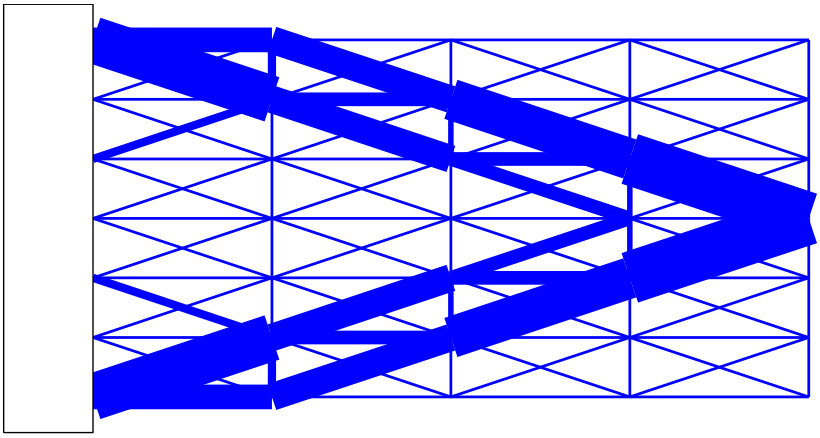
CAAM 210/211 Spring 2012

Introduction to Computational Engineering

Steven J Cox



Contents

1 **Straight Forward** **1**

2 **Loops, Matrices and Fractals** **4**

3 **Solving one Real Equation via Bisection** **8**

4 **Solving one Real Equation via Newton's Method** **11**

5 **Solving one Complex Equation via Newton's Method** **14**

6 **Boolean Gene Networks** **19**

7 **Probabilistic Boolean Gene Networks** **24**

8 **Stochastic Simulation of Reaction Networks** **30**

9 **Deterministic Chemical Kinetics** **38**

10 **Modeling of Fiber Networks** **44**

11 **Gaussian Elimination** **51**

12 **Optimal Design of Fiber Nets** **58**

13 **Multi-Layer Perceptrons** **64**

14 **Hopfield Nets** **70**

15 **Evolutionary Game Theory** **74**

1. Straight Forward

Goals: We strive for MATLAB fluency on 4 planes; To speak the language free of syntatic error, to confidently translate English simulation and/or design questions, to identify the proper numerical method for the job, and to represent the solution in the most visually striking way possible.

Means and Evaluation: We believe these goals are best attained by immersion, i.e., by regular exposure to fluent MATLAB speakers in a number of distinct engineering settings. Though distinct, these settings will all reduce to problems of network simulation or design and will introduce you to a number of important numerical methods. The fluent speakers are the **Rice Learning Assistants** that will be conducting the numerous small sections of CAAM 211. In order to bring **you** to fluency we have constructed weekly two part MATLAB excursions.

The **first part** is a short Owlspace Quiz over the science, math and programming constructs of the week. These will be "multiple choice," not timed, and in "working" each quiz you may consult written and electronic media but you may **not** discuss the quiz with any living person. They are due early each Friday and will be reviewed in your Friday lab. The quiz and lab are stuctured in order to facilitate your solution of the weekly project.

The **second part** is a full MATLAB project where you will be asked to translate English instructions into a MATLAB program that produces visual output. For two Assignments you will pledge that you received no aid in their completion. For the other assignments I encourage you to compose and troubleshoot with one another, but require you to write, document, festoon and submit your own work. By document I mean that you write your own English preamble (header) and that you write your own English comments throughout the code. By festoon I mean that you must label axes and title plots in an original and creative fashion. The main header **must** conform to this standard:

```
First Name Last Name, CAAM 210, Fall 2010, HW N
file name
program description
program usage instruction
program usage example
```

Here is one example, we shall see many more.

While you are encouraged to work with other students **currently registered** in CAAM 210, you are not allowed to consult with students who have taken previous offerings of CAAM 210 for assistance with homework assignments (except the 211 instructors and designeted college labbies). Use of external tutors must be cleared through your 210 instructor. The appearance of code obviously originating from previous semesters is considered to be a serious honor code violation.

You will **publish** each assignment to a single pdf file that you will subsequently upload to our Owlspace page. I will now explain how to publish.

- Create the directory **CAAM210**.
- Save this mfile (as type trigplot.m) to your CAAM210 directory.
- Fire up MATLAB, and set your current directory to CAAM210, by typing "cd CAAM210" at the MATLAB prompt.
- Type "edit trigplot"
- From the Editor Window click on "File" and select "Publish Configuration for trigplot.m" and follow the menu down and right to "Edit Publish Configuration for trigplot.m"
- This will bring up an "Edit M-File Configurations" window. Look for "Output settings" at about midwindow. Click and hold on the "html" as "Output file format" then drag down and select "pdf" or "doc" if you are using an older version of MATLAB.
- In the "Output folder" row please type "CAAM210" (without quotes)
- Finally click on "Publish" at the bottom right and check for the appearance of trigplot.pdf in your CAAM210 directory.

The pdf will contain a copy of your code, a list of things written to the command window, and each of the figures your code generated. If you are not happy with your pdf please close the pdf reader before trying again.

Grades: Your grades will be determined solely on the basis of the 14 quizzes, the 14 assignments and your lab participation.
211: Your score for CAAM 211 will be $(Q+P)/2$ where Q is your average quiz score (out of 100) and $P = 100a/12$ where a is the number of the 13 Friday lab sessions that you attended. Absences may be excused and late quizzes may be

accepted only with written confirmation by your doctor, academic advisor or college master.

210: Your score for CAAM 210 is

$$\frac{1}{16}\left(e_1 + e_2 + \sum_{j=1}^{14} p_j\right)$$

where p_j is your score for project j (each out of 100 except the two pledged projects which are each out of 200) and e_k is your score on the kth extra credit essay (each of which is out of 50). Late projects and essays may be accepted only with written confirmation by your doctor, academic advisor or college master.

Resources: Make use of your instructor's and/or teaching assistant's office hours. If these hours are inconvenient please ask to schedule an appointment. The graduate teaching assistant offers an optional help/review session every Thursday 7-9pm. The Friday lab is staffed by a trained Rice Learning Assistant. **It is not optional.** In the lab you will work in small groups of 3 to 5 students. The lab will not be equipped with computers. Please bring your laptop. The packet you are now reading is the textbook for this course. The definitive MATLAB resource resides online at Mathworks. We will make **constant** use of this. I strongly recommend that you take advantage of the deep student discount and purchase a copy of MATLAB from Mathworks. It costs $99 at the Mathworks Store.

2. Loops, Matrices and Fractals

To iterate is to apply the same rule over and over again. Though simple to say and easy to code this can produce some amazing patterns, e.g., eye balls, palm trees and galaxies.

To get started consider the rabbit rule 'multiply by 6'. If we start with a population of 10 and iterate we achieve 10, 60, 360, 2160, 12960, ... No, I am not amazed by this pattern. If we instead apply the cannibal rule 'multiply by 1/2' we arrive at 10, 5, 2.5, 1.25, 0.625, ... again, nothing new here.

To get more interesting patterns we move into higher dimensions. More precisely, instead of transforming one number into another via scalar multiplication we wish to transform a pair of numbers, say $x(1)$ and $x(2)$, into a second pair, say $y(1)$ and $y(2)$, via **matrix multiplication** as depicted below.

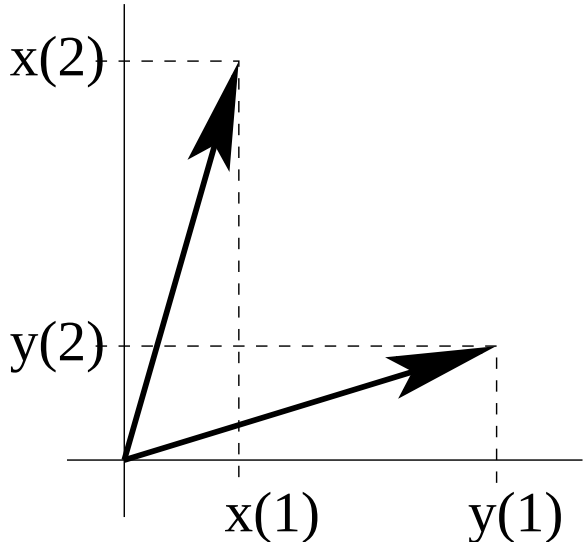


Figure 2.1. A pair of vectors in the plane.

More precisely, we write

$$y(1) = A(1,1) * x(1) + A(1,2) * x(2)$$
$$y(2) = A(2,1) * x(1) + A(2,2) * x(2)$$

or more compactly

$$y = A * x$$

where x and y are 2-by-1 vectors and A is the 2-by-2 matrix

$$A = \begin{pmatrix} A(1,1) & A(1,2) \\ A(2,1) & A(2,2) \end{pmatrix}$$

For example, the matrix

$$A = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}$$

rotates every vector clockwise by 90 degrees. Do you see it? Try it by hand for a few vectors and then coax MATLAB to do it by trying variations on the following sequence of commands:

```
>> A = [0 1; -1 0];
>> x = [1; 1];
>> y = A*x;
>> plot([0 x(1)],[0 x(2)]);
>> hold on
>> plot([0 y(1)],[0 y(2)],'r');
>> y = A*y;
>> plot([0 y(1)],[0 y(2)],'r--');
>> y = A*y;
>> plot([0 y(1)],[0 y(2)],'r-.');
>> axis equal
```

I agree that repeated application of A will just take us in circles. You yearn for something different.

Visualization in MATLAB

Although MATLAB has a friendly interactive face its power lies in its ability to chew on user generated programs. Yes, a program is just a list of legal MATLAB commands like our little example above. We will be building and changing programs of greater and greater complexity and so it is much wiser to create and save such programs than to reenter them interactively at the MATLAB command line.

One creates and saves programs through the use of a text editor. MATLAB also has a built-in editor that may be invoked it by typing **edit** from the MATLAB prompt, >>. No matter how you create any MATLAB program you should save it with a '.m' extension, e.g., caam210.m is OK. For this example, typing omg at the prompt (assuming that MATLAB's current directory contains a program omg.m) will execute the program omg.m.

Here is a program, Circle Deform I, that plots the unit circle and its deformation under a particular matrix transformation. This program waits till the end to show what it has done. To watch it work on the fly check out Circle Deform II and Circle Deform III. These programs demonstrate the use of a **for loop**. They also sport an informative **header**. They also enjoy proper **indenting** and are

sprinkled with many **comments**. They each produce a plot with an informative **title**. All of these elements, as well as the **if clause** are in evidence in our Circle Deform IV. You will use a `for` loop and an `if` clause in your first project.

I recommend that you save the Circle Deforms to your directory, run them, change the A matrix, run them, change A, run them ..., until your excitement ebbs. These runs are static incarnations of MATLAB's dynamic `eigshow` demo. I encourage you to run the demo, hit the help button, and ponder the significance of eigenvalues and eigenvectors.

An eigenvector of a matrix A is a vector that gets stretched by A but not rotated, and the eigenvalue is the amount of stretch. The geometry is clear, but is there any value in pursuing these notions? Eigen is German for self. Ask Google to Google itself by searching for **eigenvalue** and **google**. Or better yet, check out the $25,000,000,000 Eigenvector

Project: Fractal Fern

We will be growing ferns by random repetition of a few simple matrix/vector transformations. We have seen, on the lecture page, how to multiply a 2-by-2 matrix and a 2-by-1 vector and we have interpreted the "action" via stretching and rotation. A third natural geometric transformation is to shift or translate. This is done algebraically by simple vector addition. In particular, if **x**=[x(1);x(2)] and **y**=[y(1);y(2)] are two 2-by-1 vectors then their sum **z=x+y** is also 2-by-1 and its elements are z(1)=x(1)+y(1) and z(2)=x(2)+y(2).

The old-school fern below was produced by this code. You see that at each step it generates a random number and applies one of two possible transformations, the first with probability 0.3 and the second with probability 0.7.

Your task is to dissect and document this existing code and then to write and run code (called fern.m) that produces a significantly more interesting fern. In particular, your new program should successively generate a random number and should

```
apply   z = [0 0;0 0.16]*z                              with p = 0.01
apply   z = [0.85 0.04; -0.04 0.85]*z + [0; 1.6]        with p = 0.85
apply   z = [0.2 -0.26; 0.23 0.22]*z + [0; 1.6]         with p = 0.07
apply   z = [-0.15 0.28; 0.26 0.24]*z + [0; 0.44]       with p = 0.07
```

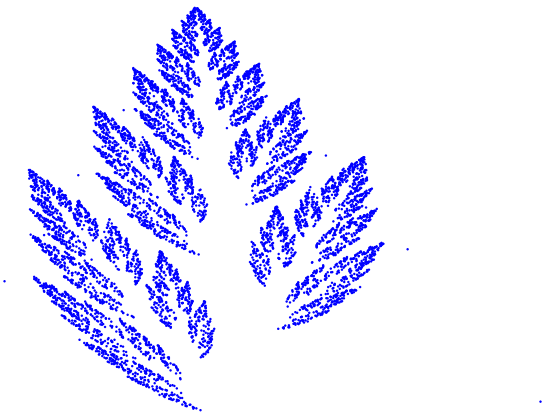


Figure 2.2. A fern assembled from two affine transformations.

You should accomplish this with a single `if` clause, although, given that your logic now forks four ways, you will want to make use of some subordinate `elseif` statements.

Once you are happy with your work please publish a single pdf via following the instructions in the Straight Forward.

Your work will be **graded** as follows:

```
1 figure:  10 pts. Use MATLAB title command to place your name on plot.

1 M-file: 10 pts for header
          10 pts for further comments in code
           5 pts for indentation in for loop
           5 pts for indentation in if clause
          10 pts for use of elseif
```

Your header must conform to the standard in the Straight Forward. For more headers and examples of inline documentation return to our circle deformers. Please recall that **each individual is solely responsible** for commenting and headering their code and for labeling their own plots.

For those who wish to dig deeper, please check out the text, **Fractals Everywhere** by M.F. Barnsley, or hit the wiki page on Iterated function system.

3. Solving one Real Equation via Bisection

Matlab provides many means with which to solve equations. The diary demonstrates the use of the polynomial root finder, **roots**, and the symbolic solver, **solve**. In response to your question, "Hey man, how do these solvers solve?" we shall devote the rest of the semester to writing Matlab solvers of increasing complexity and hence applicability.

For real scalar equations with real solutions there is a simple divide-and-conquer algorithm that is fun to visualize and easy to code. Let us take, for example, the function

$$f(x) = x^5 + x^2 - 2$$

and attempt to solve $f(x) = 0$. We start by judicious snooping and note that $f(0)f(3/2) < 0$ and so $f(0)$ and $f(3/2)$ have opposite sign and so the **Intermediate Value Theorem** permits us to conclude that a solution lies in between 0 and 3/2. To bisect now means to check the sign of $f(0)f(3/4)$ and to restrict attention to the interval [0,3/4] if negative or to [3/4,3/2] if positive or to stop if $f(3/4)$ is small enough. This process is repeated until the value of f is indeed small enough. Here are the main steps. We suppose we have a Matlab function that evaluates $f(x)$ for a given x. We also suppose the user has provided us with an interval of interest, $[a, b]$ and a tolerance t. Our job is to find an x in $[a, b]$ for which $|f(x)| < t$.

1. If $f(a)f(b) > 0$ inform the user that a solution may not exist in $[a, b]$ and exit.
2. Set $x = (a + b)/2$. While $|f(x)| > t$ do
3. If $f(a)f(x) < 0$ set $b = x$ else set $a = x$. Go to step 2.

In order to code this you shall build a function that (i) accepts the arguments, **a, b**, and **tol**, then (ii) executes steps 1-3 above and finally (iii) returns the answer.

In order to get you off on the right foot let me provide the following skeleton (it performs only 1 bisection) and a **diary** of its use

```
>> x=biskel(0,.5,1)
Sorry, but I can not be sure your fun changes sign on [a,b]
x = NaN
>> biskel(0,1.5,1)
x = 0.7500
```

I have left it for you to code steps 2 and 3. There are a number of ways to do it. I suggest a **while loop** that encloses an **if clause**.

Project: Bar Cooling

We consider a bar of length L. We suppose its temperature is zero at one end and, at the other, is proportional to the outflux of heat there. The rate at which the bar dissipates heat is the square of the least, strictly positive solution, x, of

$$\sin(xL) + x\cos(xL) = 0 \tag{3.1}$$

Our task is to determine how this root depends on the length of the bar. In particular, we will reproduce the figure below.

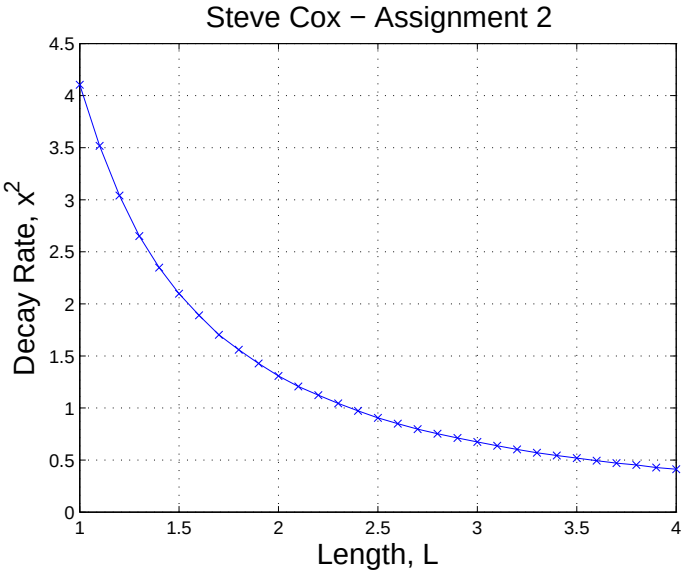


Figure 3.1. The decay rate is a decreasing function of bar length.

We will accomplish this by composing 3 simple functions within a single m-file. The m-file (called cooldrive.m) will look like

```
function cooldrive    % this is the driver
% set a, t and a range of L and b and find x
% and plot x^2 against L
return

function x = cool(a,b,t,L)  % for a given a, b, t, and L find x
% code bisection
return

function val = coolfun(x,L)  % for a given x and L evaluate "cool"
val = sin(x*L) + x*cos(x*L);
return
```

In cooldrive I would set $a = 0.1$ and $b = 3$ and $t = 0.01$ and compute the root x at each $L = 1 : .1 : 4$ and produces a figure like that above. I recommend accumulating the L and x values and issuing one plot command at the end of the loop. For example,

```
for cnt = 1:31
    L(cnt) = 1 + (cnt-1)/10;
    x(cnt) = cool(a,b,t,L(cnt));
end
plot(L,x.^2)
```

The trouble with this loop however is that (3.1) may have more than one root between $a = 0.1$ and $b = 3$. I recommend that you move one or both of these as you change L.

Your work will be graded as follows:

```
The M-file:  cooldrive.m is a function that calls cool,
             a function that calls coolfun.

   15 pts for detailed headers of cooldrive, cool, and coolfun
          featuring 'usage' and 'examples' and definitions
          of all arguments and outputs (as in biskel.m)
   10 pts for further comments in code
    5 pts for indentation
   10 pts for correct code.

Plot: 10 pts for plot of x^2 versus L labeled as above but with your name.
```

4. Solving one Real Equation via Newton's Method

Although bisection is simple and easy to code there exist far better methods (faster and applicable in more situations) for solving nonlinear equations. The best methods stem from Newton's application of what is today called

FTE : The Fundamental Trick of Engineering: Pretend the world is linear.

More precisely, suppose f is a real function of the real variable x, e.g., $f(x) = x^3 - 8$, and we wish to solve $f(x) = 0$. We begin with the hope that the solution is near some point x_0. For points x near x_0 we recall that Taylor's Theorem permits the expression

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + f''(x_0)(x - x_0)^2/2 + f'''(x_0)(x - x_0)^3/3! + \cdots$$

where $\cdots$ stands for higher order terms, meaning terms like $(x - x_0)^p$ where $p > 3$. Now Newton's Method consists in applying the FTE to the above as a means of providing a **better** guess. Namely, let x_1 be the place where the **linear part** of f near x_0 vanishes. That is, solve

$$0 = f(x_0) + f'(x_0)(x_1 - x_0)$$

for x_1. That is, set

$$x_1 = x_0 - f(x_0)/f'(x_0)$$

If f was indeed linear we would be done, i.e., f would vanish at x_1. In the nonlinear case x_1 is not a solution but is rather a (hopefully) better guess than what we started with. So you now wonder, "Can this new guess be improved?" The answer is yes. It does not require any new ideas, just stubborness. Namely, apply the FTE (repeatedly) until you arrive at a guess you deem "close enough". Anything you do repeatedly cries out to be looped. The fundamental **Newton Step** to be taken each time through the loop is

$$x_j = x_{j-1} - f(x_{j-1})/f'(x_{j-1})$$

One should exit the loop when $|f(x_j)|$ is sufficiently small.

For a slideshow graphical interpretation of each Newton Step hit this Newton Demo site.

Here is a diary of Newton's Method applied (successfully) to $f(x) = x^3 - 8$ starting from $x_0 = 1$.

Here is a diary of Newton's Method applied (unsuccessfully) to $f(x) = x^3 - 2x + 2$ starting from $x_0 = 1$.

Project: Newton vs. Bisection

Get to know this Deluxe Bisection routine. **Write** a similar Deluxe Newton routine, i.e., one whose first real line is

```
function [x, iter] = denewt(x,t,L)
```

and that also solves `coolfun(x,L)=0` for `x` when given L.
Write a driver function (`solverace`) that sets

$$L = 1, \quad a = 0.1, \quad b = 3, \quad x0 = (a+b)/2,$$

calls deluxe bisection and denewt (with starting guess $x0$), over the range of tolerances $t = 10^{-j}$, $j = 1, 2, \ldots, 8$ and produces the figure below. You will need to ask MATLAB for help with `legend` and `semilogx`.

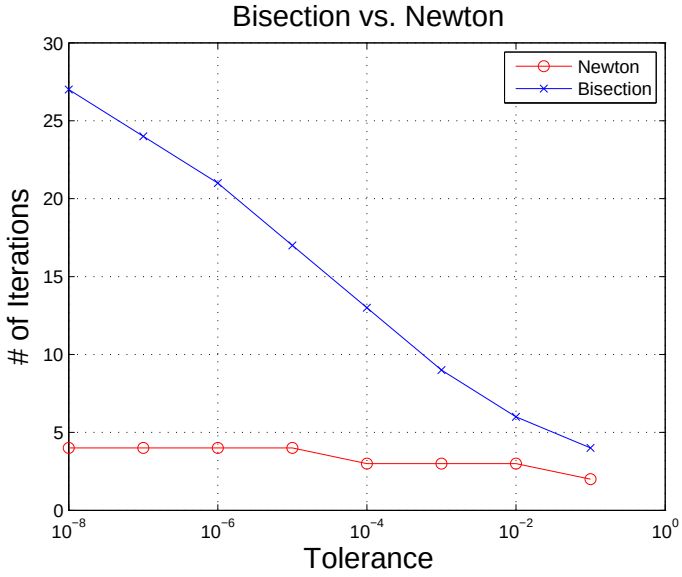


Figure 4.1. The lesser your tolerance the more you should prefer Newton.

Your code should reside in a SINGLE mfile called `solverace.m` and it should look like

```
% solverace header
function solverace
--- code --- calls debis and denewt and plots their iters per tol
return
% denewt header
function [x, iter] = denewt(x,tol,L)
--- code --- Newton's method, calls coolfun and coolfundx
return
% debis header
function [x, iter] =  debis(a,b,tol,L)
--- code --- Bisection, calls coolfun
```

```
return
% coolfun header
function val = coolfun(x,L)
--- code -- evaluate coolfun
return
% coolfundx header
function val = coolfundx(x,L)
--- code -- evaluate the derivative, with respect to x, of coolfun
return
```

Your work will be graded as follows:

```
     solverace,  6 pts for header
                 6 pts for further comments in code
                 2 pts for indentation
                 8 pts for correct code.

     denewt,     6 pts for header
                 6 pts for further comments in code
                 2 pts for indentation
                 10 pts for correct code.

     The plot,   4 points
```

5. Solving one Complex Equation via Newton's Method

We now show that Newton's method extends easily to one complex equation in one complex unknown. For example, let us see whether Newton can confirm that the roots of

$$z^2 - 2z + 2$$

are in fact

$$1 + i \quad \text{and} \quad 1 - i.$$

The Newton rule remains

$$z = z - f(z)/f'(z)$$

and so I hope it is obvious to you that a real starting guess will produce only real iterates and will not get us back to our complex roots. Please open my meandering diary and then run a few examples for yourself. Now, with a nonreal guess, our diary indeed has a happy ending. Now, if all we wanted were the roots, we would simply type `roots([1 -2 2])` at the MATLAB prompt. Our interest however, is not the destination but the journey. Along the way we will gain an appreciation for the limitation of Newton's method and we will grow our pallete of Matlab art tools.

In particular, we will dissect Isaac's attack on a polynomial by first assigning each root a color. Then, depending on where Isaac takes us we paint accordingly. For example, if Isaac takes us to the first root we paint it red, if he takes us to the other root we paint it blue, if Isaac loses his way we paint the point black. As he has a much better chance of losing his way in the complex plane we will produce a number of gorgeous Newton wastelands.

You will write code that dissects a user specfied quartic. To get started we dissect the particular

$$(z^2 - 1)(z^2 + 0.16) \tag{5.1}$$

Although the main Newton step is of course

$$z = z - (z^2 - 1)(z^2 + 0.16)/(4z^3 - 1.68z) \tag{5.2}$$

it is unclear which, if any, root this may lead to. It all depends on where we start. We shall see now that (5.2) beautifully partitions the complex plane into 5 distinct regions, four so–called **Newton Basins** and one remaining **Newton Wasteland**. The Newton Basin associated with the root $z = 1$ is defined to be all those starting points for which (5.2) eventually produces $z = 1$. The

other basins are defined accordingly. The Newton Wasteland corresponds to all those starting points for which (5.2) fails to converge to a root. For more details and examples please browse the Student Gallery and hit this Newton Basin site. With respect to our example, we choose the following color scheme

```
Basin of z=1 is yellow
Basin of z=-1 is red
Basin of z=0.4i is green
Basin of z=-0.4i is blue
Wasteland is black
```

With this scheme this pointilist driver produces the picture below

```
for x = .15:.0025:.55,
for y = -.15:.0025:.15,
    color = newt(x+i*y,1e-3,20);
    plot(x,y,color)
    hold on
end
end
```

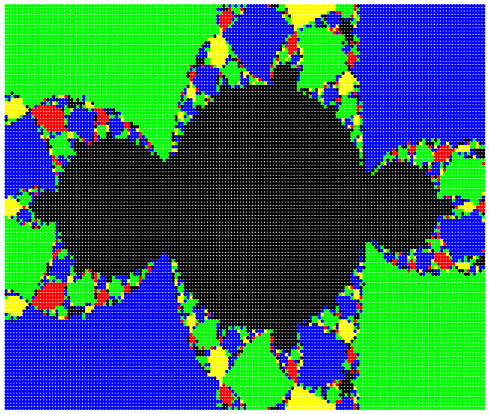


Figure 5.1. A sampling of the Newton basins of (5.1).

Let us reverse engineer what **newt** must be up to. Well, it takes a **seed** $x+iy$ and a **tolerance** (1e-3) and a **maxiter** (20) and returns a **color** ('y.', 'r.', 'b.', 'g.', or 'k.') depending on where (5.2) took the seed. The driver then paints that seed accordingly. The range of x and y specify the window of the complex plane to be painted, while the increment (0.0025) specifies the resolution. Here is the code that does the job.

Though this is indeed straightforward to understand, it does not exploit MATLAB's ability to work directly on vectors and matrices. We shall see that the nested for loops can be contracted to a **single (very fast)** line.

```
x = .15:.0025:.55;
y = -.15:.0025:.15;
[X,Y] = meshgrid(x,y);
Z = X+i*Y;
for k=1:maxiter,
    Z = Z - (Z.^2-1).*(Z.^2+0.16)./(4*Z.^3 - 1.68*Z);
end
```

Here **Z** denotes the full grid of complex Newton seed's, and, thanks to the **dot operator** (recall quad) we may operate on all seed's **simultaneously**.

Now, upon completing this for loop, **Z** will be a matrix whose elements are either close to one of the four roots, or not. To **find** out which ones are close to 1 I recommend the **find** command

```
[i1,j1] = find(abs(Z-1)<0.1);
```

Now you simply place yellow at **(x(j1),y(i1))**. In order to see what is happening please consult this 9 seed diary. If we paint the other roots accordingly, and leave the wasteland blank, we arrive at the finer (a markersize of 1) portrait below.

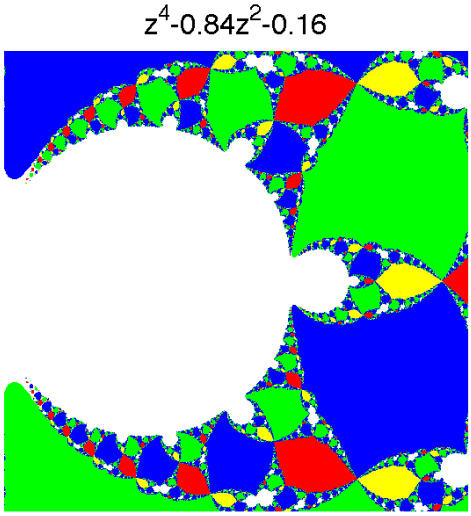


Figure 5.2. Newton basins of (5.1).

Our assignment this week is to produce such portraits for arbitrary quartics. Quartics are encoded by 5 complex scalars, for example, the quartic

$$2z^4 - 4z^3 + (2-i)z^2 - iz + 10$$

is simply

```
q = [2 -4 2-i -i 10]
```

In order to evaluate a polynomial at a given **z** one just types

```
f = polyval(q,z)
```

Now, the derivative of the polynomial above is simply

```
dq = [8 -12 4-2i -i]
```

And to evaluate it at a given **z** one just types

```
df = polyval(dq,z)
```

You may use **polyval** for this assignment but I ask that you write your own **polyder**. You should give it a new name and test it against **polyder**.

The last innovation for this assignment is the translation of the **q** into a string that will be used to title your portrait. Typing **help strfun** will lead to hours of string fun - after which you'll know all you need for this week. But just to be sure, the line

```
qlab = strcat(qlab,'+',num2str(q(k)),'z^',num2str(5-k))
```

ought to be helpful. Of course indiscriminate use of this will lead to titles like

$$1z^4 + 0z^3 + -0.84z^2 + 0z + -0.16z^0$$

rather than the slick one that adorns our finer portrait. I know I can count on you to suppress leading 1s, kill terms with leading 0s, and never write +- or z^0. You might use **strrep** to replace these unsightly strings.

Project: Newton Basins

Write a function called `qnewt` that takes as arguments

```
q,    a vector of 5 complex coefficients of a quartic
xt,   a vector of 3 x grid values, xlo, xinc and xhi
yt,   a vector of 3 y grid values, ylo, yinc and yhi
maxiter,      the max number of iterations
```

and then paints the portion of the complex plane (where $xlo \leq x \leq xhi$ and $ylo \leq y \leq yhi$) with points colored by root. Your `qnewt` must follow the `meshgrid` and `find` procedure sketched above. In using `find` you will need

to ask which elemenents of `Z` are close to the kth root. The kth root is simply `r(k)` if you have executed `r=roots(q)`.

Your qnewt must also make use of your homegrown `polyder` function. Your qnewt must automatically label its portrait with the user defined polynomial, as in Figure 5.2, achieved via

```
qnewt([1 0 -0.84 0 -0.16],[.45 .0001 .55],[-.05 .0001 .05],20)
```

Drive this code with a driver, called qdrive, that sets the grid parameters as in the example above, and calls qnewt on the above quartic and its 3 neighbors

```
[1 0 -0.84 -0.1 -0.16]
[1 -0.1 -0.84 0 -0.16]
[1 -0.1i -0.84 0 -0.16]
```

Your work will be graded as follows

```
10 pts for headers
8 pts for further comments in code
6 pts for indentation
10 pts for correct myownpolyder
10 pts for correct qlab (no 1s or 0s or +-)

8 pts for 4 plots, each titled with its associated quartic.
```

6. Boolean Gene Networks

Newton's Method can be viewed as a Discrete Dynamical System in that at prescribed times (corresponding to iterations) we move from point to point in the complex plane under the action of a specific rule, $z = F(z)$ where $F(z) = z - f(z)/f'(z)$ is a rational function built from the complex polynomial f. In fact, what we have done is to consider

$$z,\ F(z),\ F(F(z)),\ F(F(F(z))),\ \ldots$$

either until a new iterate produces no change, because $f(z) \approx 0$, or until we reach a preordained maximum number of iterations. We shall, in this chapter, widen our scope beyond the complex rational F of the last chapter. Now z will be the **state** of a gene network and F will implement the inexorable logic of gene regulation.

Our first goal will be to model and study networks of genes along the lines of the work of Wuensche. We shall assume that a gene is either on (1) or off (0) and that a gene network is merely a Boolean (after George Boole) Network, i.e., a List of nodes, with a wiring list and a logic rule for each node.

It will be convenient to assume that each gene is regulated by exactly three of its neighbors. As such there are then 8 possible inputs at each gene and so $2^8 = 256$ possible logic rules at each gene. For example, the 6 node net associated with Fig 12 in Wuensche is

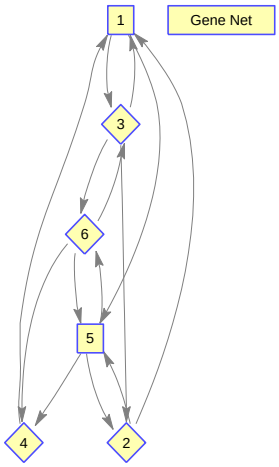


Figure 6.1. A network of six genes where each gene is regulated by three others.

This was produced via the network visualization command `biograph`, included in the Bioinformatics Toolbox, from information specified in the **wire** matrix.

```
wire = [4 2 3; 5 3 2; 3 6 1; 5 4 6; 6 1 2; 3 5 6];
n = size(wire,1);
a = zeros(n+1);
for i=1:n
    a(wire(i,:),i) = 1;
    ids{i} = num2str(i);
end
ids{n+1} = 'Gene Net';
g = biograph(a,ids);
selfcon = find(diag(a)==1);
for i=1:length(selfcon)
    g.nodes(selfcon(i)).Shape = 'diamond';
end
view(g)
```

As `biograph` shuns self-regulation we have adopted the convention that such genes will be depicted as diamonds. We next express the **action** at each gene via a table that dictates the local logic,

Table 6.1

111	110	101	100	011	010	001	000	rule
1	1	1	0	0	1	1	1	231
0	1	0	0	0	0	0	0	64
0	0	0	0	0	1	0	1	5
0	1	1	0	1	1	0	0	108
0	0	1	1	1	1	0	1	61
0	0	1	1	1	1	1	0	62

Several of these logic functions are fairly complex while several are pretty simple. For example, gene 2, inhibits itself and requires activation from both genes 5 and 3. In other words, if $s(i)$ and $ns(i)$ are the respective current and subsequent (next) states of gene i then

$$ns(2) = (s(5) \text{ AND } s(3)) \text{ AND } (\text{NOT } s(2)).$$

Similarly, the logic at gene 6 follows

$$ns(6) = (s(3) \text{ XOR } s(5)) \text{ OR } (s(6) \text{ AND NOT } (s(3) \text{ OR } s(5))).$$

Let us compute a number of exact cases. For example, if

```
s = (1 0 1 0 0 1)
```

then upon consulting `wire(1,:)` and `s` we find that 001 is the pattern presented to gene 1. Consulting the associated column in row 1 of Table 6.1 we conclude that $ns(1) = 1$.

Next, on consulting `wire(2,:)` and `s` we find that 010 is the pattern presented to gene 2. Consulting the associated column in row 2 of Table 6.1 we conclude that $ns(2) = 0$.

Next, on consulting `wire(3,:)` and `s` we find that 111 is the pattern presented to gene 3. Consulting the associated column in row 3 of Table 6.1 we conclude that $ns(3) = 0$.

This is indeed very mechanical and so is best turned over to MATLAB. In this case we arrive at

```
ns = (1 0 0 0 0 1).
```

As it happens `ns` is also the next state of itself and so is called a **point attractor** of the gene net described by Table 6.1.

We now come to the question of how best to **diagram** the transitions between each of the 2^n states of a gene net with n genes. We begin by numbering the states from 0 to $2^n - 1$ by equating them with there decimal equivalents, e.g.,

```
(1 0 1 0 0 1) = 2^5 + 2^3 + 2^0 = 41  and
(1 0 0 0 0 1) = 2^5 + 2^0 = 33.
```

We then construct a 2^n-by-2^n State Transitition Matrix, `STM`, of zeros, with a single 1 in each row, where `STM(i,j)=1` if state `i-1` transitions to state `j-1`. (We subtract 1 because MATLAB row and column numbers begin with 1, not 0). Once `STM` is built we may hand it to `biograph` to produce the associated State Transition diagram.

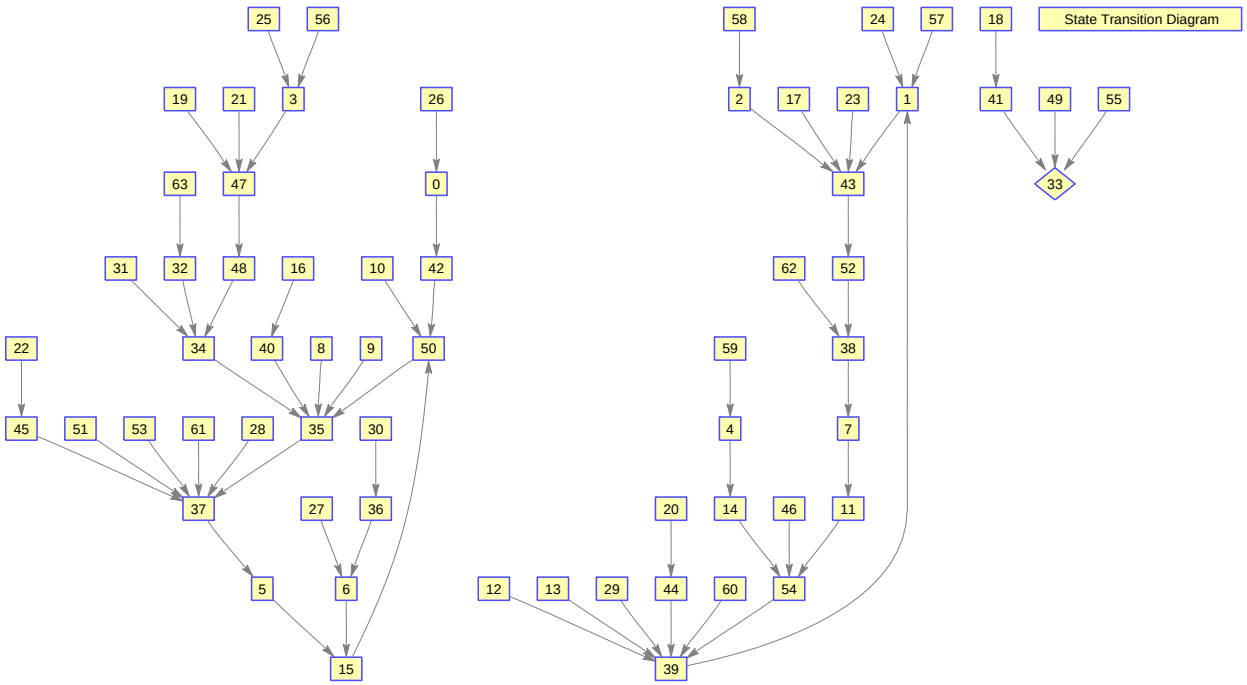


Figure 6.2. The State Transition Diagram for the net specified in Table 6.1.

We see indeed that state $41 \to 33 \to 33$ as computed above. We also see two (more fascinating) basins of attraction associated with the 5-state attractor

$$50 \to 35 \to 37 \to 5 \to 15 \to 50$$

and the 8-state attractor

$$1 \to 43 \to 52 \to 38 \to 7 \to 11 \to 54 \to 39 \to 1.$$

In this week's project we will investigate the sensitivity of these attractors to mutations in the logic rules.

Project: Gene Networks I

Write a

```
function STM = genestm(wire,rule)
```

that accepts an n-by-3 wire matrix and an n-by-1 rule vector and returns the associated State Transition Matrix.

After determining n, the number of genes, I would translate the rule vector into an n-by-8 rule matrix (without the top and right borders) like that in Table 6.1. I would do this one row at a time via a subfunction of the form

```
function b = d2b(r,C)
```

that converts decimal numbers to C-bit binary. I ask that you **not use** the builtin functions bin2dec and dec2bin, for they are too slick for us. **Hint:**

```
b(C-floor(log2(r)))=1,  reset r,  repeat.
```

In order to build `STM` I would proceed like

```
for i=1:2^n
    s = d2b(i-1,n);
    ns = next state (using s, wire and rulemat)
    j = b2d(ns);   your binary to decimal converter
    STM(i,j+1) = 1;
end
```

Finally, include your `genestm` function under a function called `genestmdriver` that takes no arguments, but sets wire and rule (to the values used in our example above), draws the gene net using `biograph` as above, calls your `genestm` and plots your State Transition Diagram. Next, flip one bit in your rule vector, e.g., `rule(2)=192` or `rule(4)=44`, and repeat all of the above steps.

Your work will be graded as follows:

```
10 pts for headers for genestmdriver, genestm, d2b and b2d
8 pts for further comments in code
4 pts for indentation
8 pts for correct rule matrix (displayed to command window)
8 pts for correct computation of next state
4 pts for correct STM

2 pts for one gene net plot
6 pts for two State Transition Diagrams
```

7. Probabilistic Boolean Gene Networks

Our experimental understanding of the logic of gene regulation is rarely (if ever) as clean as that presented in the last section. It is more often the case that regulation follows multiple possible courses. Probabilistic Boolean Networks (PBN) were invented to deal with this scenario. In this case, each gene i has $R(i)$ rules

$$r(i,1),\ r(i,2),\ldots,r(i,R(i))$$

and associated probabilities

$$p(i,1),p(i,2),\ldots,p(i,R(i)).$$

For example, lets consider the fully coupled 3-gene net

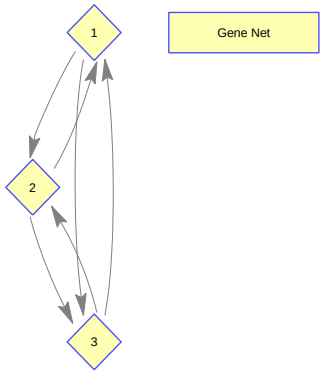


Figure 7.1. A network of 3 genes with 3-way regulation.

with wire, rules and probabilities

Table 7.1

gene	wire	rule	probability
1	1 2 3	238, 230	3/5, 2/5
2	1 2 3	64	1
3	1 2 3	232, 128	1/2, 1/2

As the number of rule choices varies with each gene we need a new data structure for `rule`. We recommend the `cell`, e.g.,

```
rule = {[238 230]
        [182]
        [232 128]};
```

With n genes we now have

$$N = \prod_{i=1}^{n} R(i)$$

possible (old school, nonprobabilistic) gene networks. In our example case we depict the rule tree that leads to these $N = 4$ gene nets.

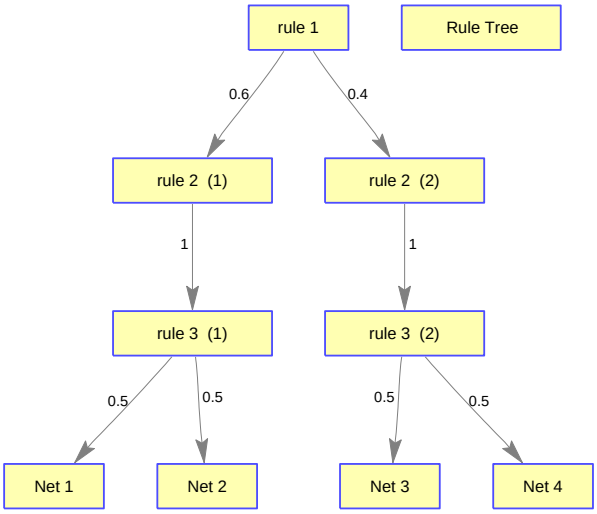


Figure 7.2 The rule tree for the PBN specified in Table 7.1. The branch probabilities were displayed via `biograph(ptree,ids,'showweights','on')`.

We begin with rule 1 at the top (root) of the tree and draw two branches (with probabilities) associated with the 2 rule choices. At the next generation we choose the rule 2 with probability 1. As rule 3 has two possiblities we again branch at each node and arrive at 4 possible nonprobabilistic gene nets. Let us develop this in somewhat great detail. In particular, let us build `ptree`, the adjacency matrix that we hand to `biograph` that then illustrates our rule tree.

There is first the question of size. How many nodes are there in the rule tree? The numerical answer is `Ntree = 1 + 2 + 2 + 4`. How can you build this number automatically from the `R` vector, where `R(i)` = number of elements of `rule{i}`. Once `Ntree` is found we can initialize via `ptree = zeros(Ntree+1)` and label the first node via `ids{1} = 'rule 1'`.

We next specify the first row of `ptree` by "connecting" the top node to the next two nodes with the proper weights

```
ptree(1,2) = prob{1}(1)
ptree(1,3) = prob{1}(2)
```

or, in one line, via `ptree(1,2:1+R(1)) = prob{1}`. Next we set their

associated labels

```
ids{2} = 'rule 2 (1)'
ids{3} = 'rule 2 (2)'.
```

We next connect these nodes to the next level of the tree via

```
ptree(2,4) = prob{2}    ptree(3,5) = prob{2}
```

and set their associated labels via

```
ids{4} = 'rule 3 (1)'    ids{5} = 'rule 3 (2)'.
```

Finally we connect to the last level via

```
ptree(4,6) = prob{3}(1)    ptree(4,7) = prob{3}(2)
ptree(5,8) = prob{3}(1)    ptree(5,9) = prob{3}(2),
```

and set their labels via

```
ids{6} = 'Net 1'   ids{7} = 'Net 2',
ids{8} = 'Net 3'   ids{9} = 'Net 4'.
```

This results in the `ptree` matrix

$$\begin{pmatrix} 0 & 0.6 & 0.4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.5 & 0.5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.5 & 0.5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Your challenge is to automate the construction of this matrix, and to assemble the associated probabilities and rule vectors of the final `N` nets.

The probability, $P(i)$, of arriving at the ith net is simply the product of the branch probabilities that connect the ith leaf to the root. For example, the net probabilities for the PBN specified in Table 7.1. are

$$\begin{aligned} P(1) &= \texttt{prob\{1\}(1)prob\{2\}(1)prob\{3\}(1)} = 3/10 \\ P(2) &= \texttt{prob\{1\}(1)prob\{2\}(1)prob\{3\}(2)} = 3/10 \\ P(3) &= \texttt{prob\{1\}(2)prob\{2\}(1)prob\{3\}(1)} = 2/10 \\ P(4) &= \texttt{prob\{1\}(2)prob\{2\}(1)prob\{3\}(2)} = 2/10. \end{aligned}$$

For the ith net we may compute the State Transition Matrix, STM_i, as in the

previous chapter. The full Probabilistic State Transition Matrix is then simply

$$PSTM = \sum_{i=1}^{N} P(i) STM_i. \tag{7.1}$$

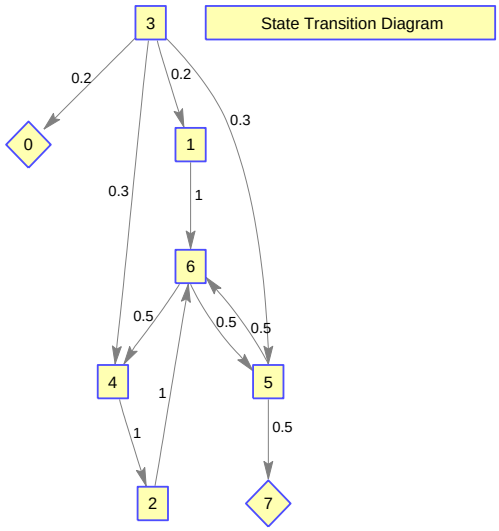


Figure 7.3. The State Transition Diagram for the PBN specified in Table 7.1.

We observe that states 0 and 7 are point attractors while $6 \to 4 \to 2 \to 6$ and $6 \to 5 \to 6$ are pseudo–attractors.

Project: Probabilistic Boolean Gene Networks

Write a

```
function [pnet, rnet] = ruletree(rule,prob)
```

that accepts rule and prob cells, plots the associated rule tree (see Figure 7.4) and returns a vector, `pnet`, of N net probabilities and a vector, `rnet`, of N rule indicators.

Append this function to a function called `pbndriver` that takes no argument but sets `wire, rule` and `prob` to

```
wire = [4 2 3;5 3 2;3 6 1;5 4 6;6 1 2;3 5 6];
rule = {[231 230], 64, [5 7], 108, 61, [62 60]};
prob = {[1/2 1/2], 1, [1/2 1/2], 1, 1, [1/2 1/2]};
```

calls `ruletree` and proceeds to build the Probabilistic State Transisition Matrix of (7.1) by using last week's code to compute the State Transition Matrices

of the N oldschool nets encoded in `rnet`. Finally, use `biograph` to depict the final State Transition Diagram as in Figure 7.5.

Regarding the format of `rnet` I recommend using powers of 10. In this case, the output of `ruletree` should be

```
pnet = 0.1250 0.1250 0.1250 0.1250 0.1250 0.1250 0.1250 0.1250
rnet = 111111 111112 112111 112112 211111 211112 212111 212112
```

in which case we see that each net is equally likely, that the first net uses the first rule at every gene, that the second net uses the first rule at gene 1 through 5 and the second rule at gene 6, *etc.*

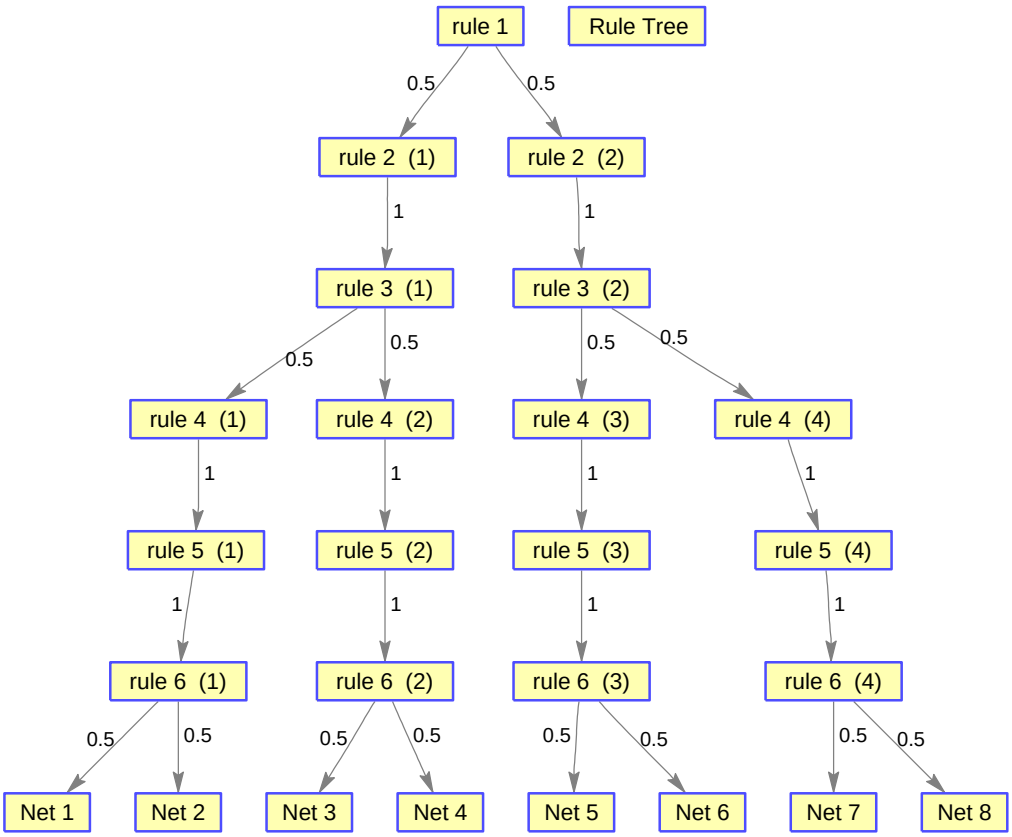


Figure 7.4. Rule Tree for the PBN specified above.

Your work will be graded as follows:

```
10 pts for headers for pbndriver and ruletree
8 pts for further comments in code
4 pts for indentation
8 pts for correct pnet computation
8 pts for correct rnet computation
```

4 pts for correct assembly of PSTM

4 pts for ruletree plot
4 pts for State Transition Diagram

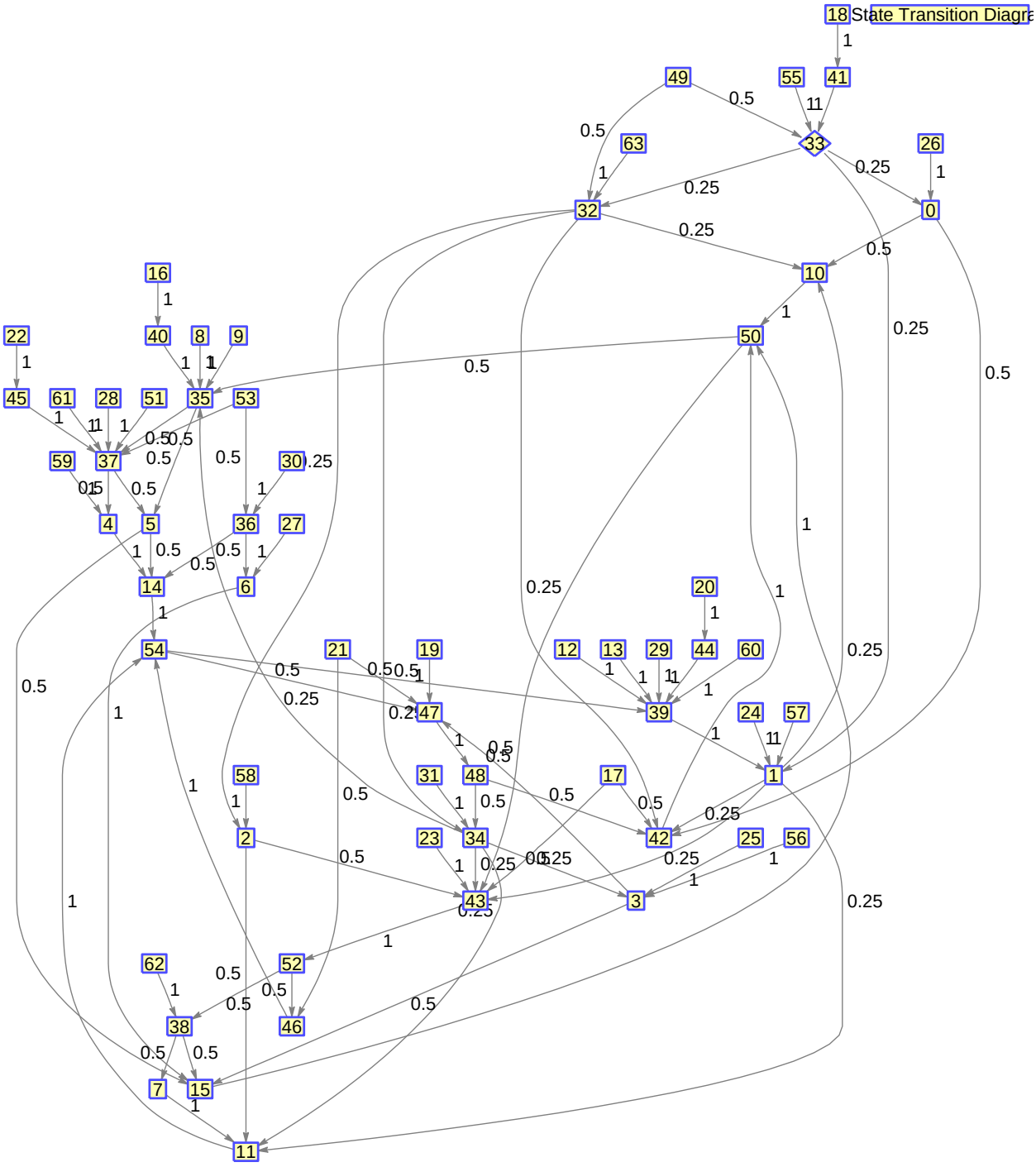


Figure 7.5. The Probabilistic State Transition Diagram.

8. Stochastic Simulation of Reaction Networks

Our Boolean approach to gene regulation invoked the **logic** rule of each operon without questioning the underlying biophysical mechanism. The key players are the gene (stretch of DNA) and its reader (RNA Polymerase). Governing genes regulate the transcription of their subjects by either promoting or prohibiting the binding of RNAP to the subject's stretch of DNA.

This suggests that we might benefit from a tool that simulates the interactions that occur in a network of reacting chemical species. In the next two chapter we will consider two standard means for modeling and simulating such systems. The first is associated with the name of Gillespie and the second with the names of Michaelis and Menten.

Stochastic Chemical Kinetics

This is a beautifully simple procedure for capturing the dynamics (kinetics) of a randomly interacting (stochastic) bag of chemicals (us?).

We begin with N reacting chemical species, $S_1, S_2, \ldots, S_N$, and their initial quantities

$$X_1, X_2, ..., X_N$$

We suppose that these species interact via M distinct reactions

$$R_1, R_2, ..., R_M$$

and that these reactions occur with individual **propensities**.

$$c_1, c_2, ..., c_M$$

For example, if R_1 is

$$X_1 + X_2 \to X_3$$

then we suppose that the average probability that a particular S_1S_2 pair react within time dt is c_1dt. If, at time t, there are X_1 molecules of S_1 and X_2 molecules of S_2 then there are X_1X_2 possible pairs and hence the probabilty of reaction R_1 occuring in the window $(t, t + dt)$ is $X_1X_2c_1dt$. (Under the assumption that molecules of S_j are hard spheres of diameter d_j and mass m_j, and that the reaction takes place in region of volume, V, and temperature, θ, only when the associated kinetic energy of the colliding spheres exceeds the activation energy,

u_1^*, it follows that the propensity

$$c_1 = \frac{\pi}{V}\left(\frac{d_1+d_2}{2}\right)^2\sqrt{\frac{8k\theta(m_1+m_2)}{\pi m_1 m_2}}\exp(-u_1^*/(k\theta)),$$

where k is Boltzmann's constant).

For more general reactions we will write h_m for the number of distinct R_m molecular reactant combinations available in the state $(X_1, ..., X_n)$. We then arrive at the reaction probability $a_m = h_m c_m$. We shall have occasion to call on

$$a_0 = a_1 + a_2 + \cdots + a_M$$

Given this list of reactions and their propensities one naturally asks

Which reaction is likely to happen next?

and

When is it likely to occurr?

Daniel Gillespie answered both questions at once by calculating

$$P(T, m)dT,$$

the probability that, given the state $(X_1, ..., X_n)$ at time t, the **next** reaction will be reaction m **and** it will occur in the interval $(t + T, t + T + dT)$.

A careful reading permits us to write $P(T, m)dT$ as the product of two more elementary terms, namely, the probability that **no** reaction occurs in the window $(t, t + T)$, and the probability that reaction m occurs within time dT of T. We have already seen the latter, and so, denoting the former by $P_0(T)$, we find that

$$P(T, m)dT = P_0(T)a_m dT$$

Regarding $P_0(T)$, as the probabilty that **no** reaction will occur in $(t, t + dT)$ is $1 - a_0 dT$ it follows that

$$P_0(T + dT) = P_0(T)(1 - a_0 dT)$$

or

$$(P_0(T + dT) - P_0(T))/dT = -a_0 P_0(T)$$

and which, in the limit of small dT, states

$$P_0'(T) = -a_0 P_0(T)$$

and so

$$P_0(T) = \exp(-a_0 T)$$

It now comes down to drawing or "generating" a pair (T, m) from the set of random pairs whose probablity distribution is

$$P(T, m) = a_m \exp(-a_0 T)$$

Gillespie argues that this is equivalent to drawing two random numbers, r_1 and r_2, from the [0,1]-uniform distribution and then solving

$$r_1 = \exp(-a_0 T) \tag{8.1}$$

for T and solving

$$a_1 + a_2 + ... + a_{m-1} < r_2 a_0 < a_1 + a_2 + ... + a_m \tag{8.2}$$

for m. We have now assembled all of the ingredients of Gillespie's original algorithm:

```
Gather propensities, reactions, initial molecular
counts, and maxiter from user/driver.
Set t = 0 and iter = 0;

while iter < maxiter
      Plot each X_j at time t
      Calculate each a_j
      Generate r_1 and r_2
      Solve (8.1) and (8.2) for T and m
      Set t = t + T
      Adjust each X_j according to R_m
      Set iter = iter + 1
 end
```

The gathering of reactions, like the gathering of wire and rule, may take some thought. For small networks we can meet the problem head on. We reproduce two examples from Gillespie. The first is

$$\begin{aligned} X_\infty + Y &\to 2Y \qquad \text{with propensity } c_1 \\ 2Y &\to Z \qquad \text{with propensity } c_2 \end{aligned} \tag{8.3}$$

Here the subscript on X_∞ indicates that its level remains constant (either because it corresponds to a constant feed or it is in such abundance that the first reaction will hardly diminish it) throughout the simulation. Here is the code that produced the figure below.

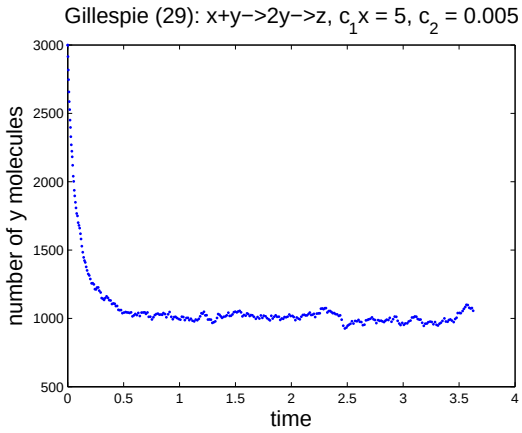


Figure 8.1. The evolution of Y following (8.3).

Though poorly documented you can nonetheless see each step of the algorithm. Our second example is

$$\begin{aligned} X_\infty + Y_1 &\to 2Y_1 \qquad \text{with propensity } c_1 \\ Y_1 + Y_2 &\to 2Y_2 \qquad \text{with propensity } c_2 \\ Y_2 &\to Z \qquad \text{with propensity } c_3 \end{aligned} \tag{8.4}$$

Roughly speaking, the Y_1 species procreates, is consumed by the Y_2 species, which itself dies off at a fixed rate. Such models are known as predator-prey and, one can imagine, that the two populations often keep one another in check.

For example, here is the code that generated the lovely figure below.

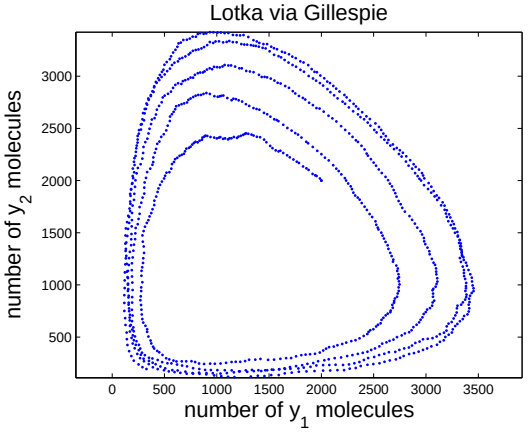


Figure 8.2. The evolution of Y_1 and Y_2 following (8.4).

Of course for larger reaction networks we, I mean you, must proceed more systematically. On the road to a more general robust method we wish to address

1. The user is unsure of how to choose maxiter and would prefer to provide the final time, tfin.

2. The user may enter the reaction list as a cell and code the update of h in a subfunction and so preclude a very lengthy if clause.

3. A single run of Gillespie is rarely sufficient - for it delivers but one possible time evolution. The user would prefer that we made nr runs and collected and presented statistics across runs.

We may incorporate the first point by exchanging Gillespie's `for` for a `while`.

To address the second point, I imagine a cell where each entry encodes a full reaction, say, e.g., by adopting the following convention

```
rtab{k}(1:2:end) = indicies of reaction species
rtab{k}(2:2:end) = actions for species above
```

for the kth reaction. In the Lotka example, this would read

```
rtab = {[1 1]         % Y1 -> Y1 + 1
        [1 -1 2 1]    % Y1 -> Y1 - 1 and Y2 -> Y2 + 1
        [2 -1]}       % Y2 -> Y2 - 1
```

and the update function would look like

```
function h = update(y)
h(1) = y(1);
h(2) = y(1)*y(2);
h(3) = y(2);
```

The use of `update` is relatively straightforward, e.g.,

$$a = c. * h;$$

With this a in hand you may now use `cumsum` and `find` to find the next reaction index (in just a couple of lines - and with NO `if` clauses). With this index in hand you may visit the proper element of `rtab`.

To address the third point above, we should suppress plotting on the fly and instead return the time and desired solution vectors to a driver that will collect and plot statistics. In particular, if we lay our Gillespie runs along the rows of a big matrix and take means (averages) down the columns, e.g.,

```
for j=1:nr
    [t,x] = mygill(tfin,rtab,x0,c);
    X(j,:) = x;
end
avg = mean(X,1);
```

we "ought" to be on the right track. The trouble is that each of our Gillespie runs are loyal to particular time vectors. One way around this is to legislate a uniform time vector

```
tvec = 0:tinc:tfin
```

and bring the individual x vectors into X only after interpolating them with `interp1`.

Project - Stochastic Operon Simulation

We will follow McAdams and Arkin in their application of Gillespie's method to a simple model of gene expression. In particular we model the binding of RNA polymerase, R, to the gene promoter segment, Pr. The complex, RPr, may either decompose into its constituents or it may be transcribed and translated in a process that creates 10 new molecules of protein, P, in addition to the free constituents, R and Pr. Each protein molecule undergoes degradation and reversible dimerization (protein–protein binding). We depict these 6 reactions via

$$\begin{aligned} R + Pr &\rightarrow RPr \quad \text{RNAP promoter binding and opening} \\ RPr &\rightarrow R + Pr \quad \text{RNAP promoter closing and unbinding} \\ RPr &\rightarrow 10P + R + Pr \quad \text{protein production} \\ P + P &\rightarrow D \quad \text{dimerization} \\ D &\rightarrow P + P \quad \text{dedimerization} \\ P &\rightarrow \quad \text{protein degradation} \end{aligned} \tag{8.5}$$

and collect the reactants into the vector

$$x = [R\ Pr\ RPr\ P\ D]$$

and follow their evolution from the initial levels

$$x0 = [10\ 1\ 0\ 0\ 0]$$

and 6 propensities

$$c = [2\ 1\ 4\ 2\ 0.5\ 0.05].$$

Working from the outside in, you will **write** a function called `gilldriver` that sets

```
tfin,   the duration of the simulation
```

```
tinc,    the increment between interpolated time points
nr,      the number of Gillespie runs
rtab,    a cell array that encodes the M reactions
x0,      the initial levels of x
c,       the propensities
```

and call your subfunction `mygill` nr times and plots (as below) the mean (blue) and plus/minus one standard deviation (red) of the dimer solutions. Your `mygill` subfunction should behave like

```
function [t,dimer] = mygill(tfin, rtab, x, k)
```

and should call on its associated `update` function and should not require any `if` clauses.

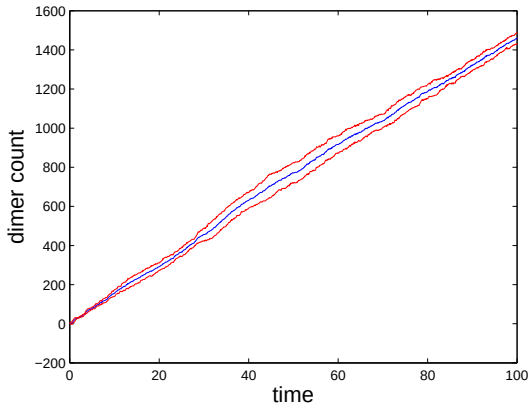


Figure 8.3. Unchecked dimer growth.

In the absense of inhibition the dimer count is roughly increaing. To complete this assignment I ask you to augment your gilldriver to incorporate repression of Pr by D via the two additional reactions

$$\begin{aligned} Pr + D &\rightarrow I \qquad \text{D binds Pr and so R can not read} \\ I &\rightarrow Pr + D \qquad \text{D unbinds Pr} \end{aligned} \tag{8.6}$$

I would start from $I = 0$ and assume propensities of 0.01 and 0.005 and produce something like the figure below. (I believe I used nr=4. You are encouraged to experiment with propensities and run numbers.)

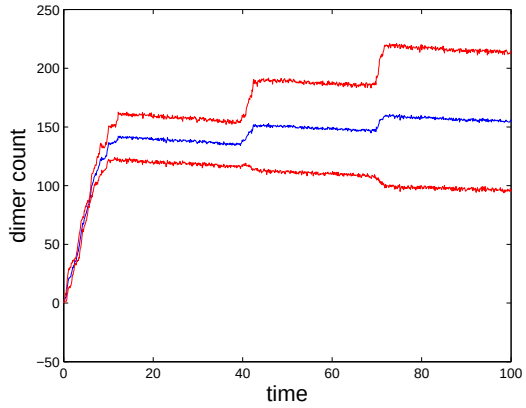


Figure 8.4. Dimer Self-control.

Your work will be graded as follows:

```
 6 pts for header CONTAINING detailed USAGE
 6 pts for further comments in code
 4 pts for indentation
10 pts for correct gilldriver
10 pts for correct computation of reaction number
 8 pts for correct coding of update

 6 pts for two plots corresponding to the free dimer
       production, and to its self-regulated production.
```

9. Deterministic Chemical Kinetics

The stochastic view is a bare-handed means of following the dynamics of individual molecules. For large reacting systems this approach may be prohibitively expensive and/or unnecessarily precise. In such cases one may rely on more coarse grained tools, e.g., the

Law of Mass Action: The rate of a chemical reaction is directly proportional to the product of the effective concentrations of each participating reactant.

We will come to understand this law by its application.

Example 1. Suppose R_1 is the bi-reactant system

$$X_1 + X_2 \to X_3$$

that proceeds with propensity c_1 in a region of volume V. It follows that there are X_1X_2 distinct combinations of reactants inside V, and so $X_1X_2c_1dt$ gives the probability that R_1 will occur somewhere inside V in the next dt units of time. If we now average over nr runs,

$$\text{mean}\,(X_1X_2c_1) = c_1\,\text{mean}\,(X_1X_2)$$

is the *average rate* at which R_1 reactions are occurring inside V. The average reaction rate per unit volume is therefore

$$\text{mean}\,(X_1X_2)c_1/V = Vc_1\text{mean}\,(x_1x_2)$$

in terms of the molecular concentrations

$$x_j \equiv X_j/V.$$

Now, the reaction rate constant, k_1, is **defined** to be this average reaction rate per unit volume divided by the product of the average concentrations of the reactants

$$k_1 = \frac{Vc_1\text{mean}\,(x_1x_2)}{\text{mean}\,(x_1)\text{mean}\,(x_2)}.$$

Finally, on equating the average of a product with the product of the averages, i.e.,

$$\text{mean}\,(x_1x_2) = \text{mean}\,(x_1)\text{mean}\,(x_2)$$

we arrive at

$$k_1 = Vc_1. \tag{9.1}$$

In this case the law of mass action dictates that

$$x_1'(t) = x_2'(t) = -k_1 x_1(t) x_2(t) \quad \text{and} \quad x_3'(t) = k_1 x_1(t) x_2(t).$$

Example 2. Suppose R_2 is the mono-reactant system

$$2X_1 \to X_3$$

that proceeds with propensity c_2 in a region of volume V. In this case there are $X_1(X_1-1)/2$ possible pairs and if X_1 is large then this number is relatively close to $X_1^2/2$. As such

$$k_2 = V c_2/2$$

and mass action dictates that

$$x_1'(t) = -2k_2 x_1^2(t) \quad \text{and} \quad x_3'(t) = k_2 x_1^2(t).$$

Example 3. We return to (8.3)

$$\begin{aligned} X_\infty + Y &\to 2Y \qquad (\text{propensity } c_1) \\ 2Y &\to Z \qquad (\text{propensity } c_2) \end{aligned}$$

and, with $k_1 = c_1 V$ and $k_2 = c_2 V/2$ find

$$\begin{aligned} y'(t) = 2k_1 x_\infty y(t) &- (k_1 x_\infty y(t) + 2k_2 y^2(t)) \\ \text{gains} \qquad &- \qquad \text{losses} \end{aligned} \tag{9.2}$$

If we denote the initial concentration $y(0) = y0$ we may solve the ordinary differential equation (9.2) "by hand" (via calculus or

```
dsolve('Dy = k1x*y-2*k2*y^2','y(0)=y0')
```

in MATLAB) and find

$$y(t) = \frac{y_0 k_1 x_\infty}{2y_0 k_2 - (2y_0 k_2 - k_1 x_\infty)\exp(-k_1 x_\infty t)}$$

This function is easy to graph and so compare with its stochastic cousin. Assuming $V = 1$ we arrive at the figure below on running gill1o.m

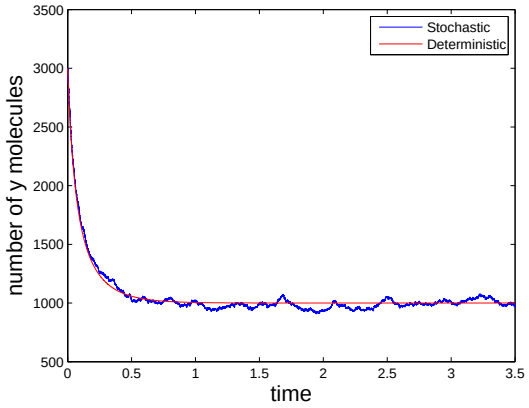


Figure 9.1. Stochastic *vs.* Deterministic approach to (8.3).

We see that the deterministic model does an excellent job of tracking the stochastic trajectory.

Example 4. The Lotka system

$$\begin{aligned} X_\infty + Y_1 &\to 2Y_1 \qquad \text{propensity } c_1 \\ Y_1 + Y_2 &\to 2Y_2 \qquad \text{propensity } c_2 \\ Y_2 &\to Z \qquad \text{propensity} c_3 \end{aligned}$$

yields, with $k_j = c_j V$,

$$\begin{aligned} y_1' &= k_1 x_\infty y_1 - k_2 y_1 y_2 \\ y_2' &= k_2 y_1 y_2 - k_3 y_2 \end{aligned}$$

The solution of this system is not presentable in "closed form" and so we turn to approximate, or numerical, means. MATLAB sports an entire suite of programs for solving differential equations. The default program is `ode23`. We invoke it in our Lotka 2-way code and arrive at the figure below. As above blue is the Gillespie trajectory and red is the ode trajectory.

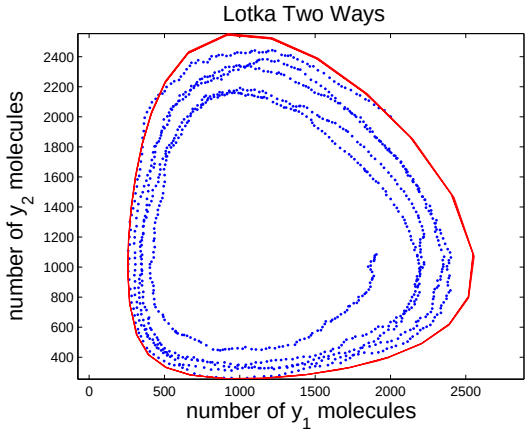


Figure 9.2. Stochastic *vs.* Deterministic approach to (8.4).

Example 5. The Brusselator system

$$\begin{aligned} X_{\infty,1} &\to Y_1 \qquad \text{propensity } c_1 \\ X_{\infty,2} + Y_1 &\to Y_2 + Z_1 \qquad \text{propensity } c_2 \\ 2Y_1 + Y_2 &\to 3Y_1 \qquad \text{propensity } c_3 \\ Y_1 &\to Z_2 \qquad \text{propensity } c_4 \end{aligned}$$

becomes

$$\begin{aligned} y_1' &= c_1 x_{\infty,1} - c_2 x_{\infty,2} y_1 + (3-2)(c_3/2) y_1^2 y_2 - c_4 y_1 \\ y_2' &= c_2 x_{\infty,2} y_1 - (c_3/2) y_1^2 y_2 \end{aligned} \tag{9.3}$$

This associated ode code produces the figures below. You may wish to zoom in to better see the transitions that each species undergoes, or better yet, take a look at the (y_1, y_2) plane.

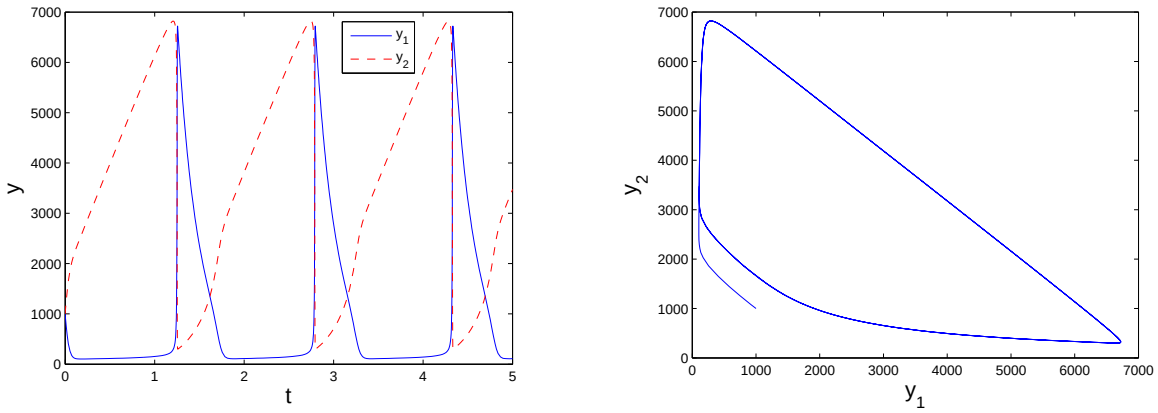


Figure 9.3. Stochastic *vs.* Deterministic approach to (9.3).

Deterministic *vs.* Stochastic Operon Simulation

We continue last week's investigation of a self-governing gene. Recall the 8 reactions in (8.5) and (8.6) involved the 6 species

$$X = [R \; Pr \; RPr \; P \; D \; I]$$

and propensities c(1) throught c(8).

Your first task is to write down the associated 6 ordinary differential equations (in the order they are listed in x above). I will give you the first one,

$$x_1' = -c_1 x_1 x_2 + c_2 x_3 + c_3 x_3$$

Your next task is to augment last week's code so that it solves (via `ode23`) this ode system and compares the mean (stochastic) dimer level to the ode

prediction. For example, using the initial counts and propensities

$$x0 = [10\ 1\ 0\ 0\ 0\ 0] \quad \text{and} \quad c = [2\ 1\ 4\ 2\ 0.5\ 0.05\ .01\ .005]$$

and $nr = 10$, we arrive at the figure below

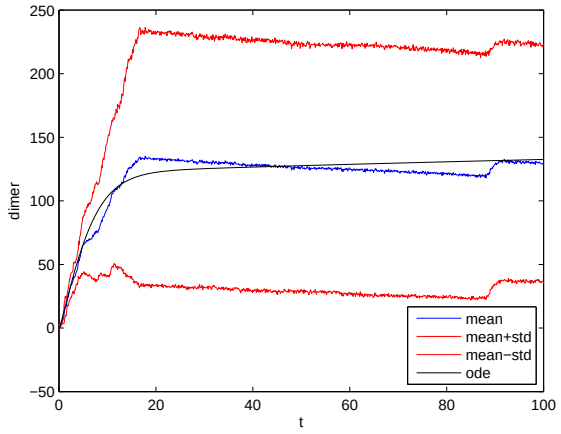


Figure 9.4. Average Stochastic *vs.* Deterministic approach to the self-regulating gene.

This result is promising, in that the dimer predictions appear to coincide. It would be nice to dig a little deeper and ask how the other reactants are fairing. At the stochastic level we know that each reactant may only take integer jumps and that R, Pr, RPr and I in fact jump back and forth between 0 and 1. As taking ensemble means introduces intermediate values we instead compare our ode solution to single runs of Gillespie. The figure below was obtained with the same initial values and propensities as above.

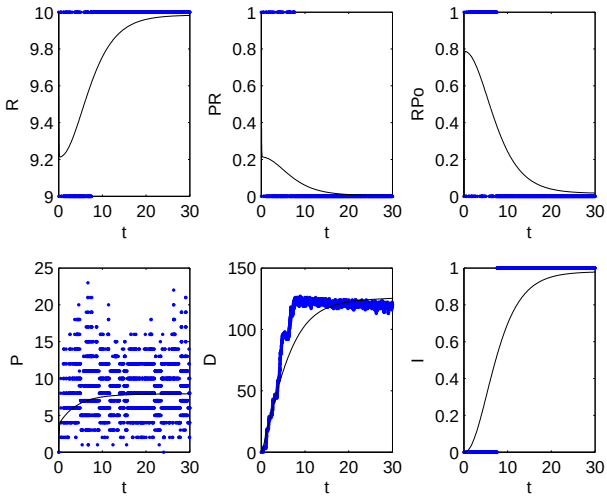


Figure 9.5. Solo Stochastic *vs.* Deterministic approach to the self-regulating gene.

This figures cries out for explication. It shows early switching in R, Pr and RPr, associated with stable early growth in D, despite the high volatility in P. And finally, once D achieves some critical level we see I switch on and stay on.

You will accomplish these two figures by changing last week's gilldriver to a function called mca2driver that sets sets tfin, tinc, nr, x0, and c (first with nr=4 or so), calls mygill nr times and plots the mean and standard deviation like above, then calls ode23 which in turn calls

```
function dx = mcaode(t,x,c)
```

and then plots the new dimer count as in our first figure above. Your driver then sets $nr = 1$ and constructs, via `subplot(2,3,j)`, the lovely six panel figure above. In order to "see" through the noise I have limited the width of this window relative to the tfin used above. You too can do this with `xlim([0 30])`.

Your work will be graded as follows:

```
6 pts for header CONTAINING detailed USAGE
      header of mcaode should give full system in detail
6 pts for further comments in code
4 pts for indentation
14 pts for correct mca2driver
10 pts for correct mcaode
10 pts for two plots similar to those above
```

10. Modeling of Fiber Networks

We consider the case illustrated in Figure 10.1. The bold solid line is a fiber in its reference state. When we subject its two ends to the two forces, (f_1, f_2) and (f_3, f_4) the respective ends are displaced by (x_1, x_2) and (x_3, x_4).

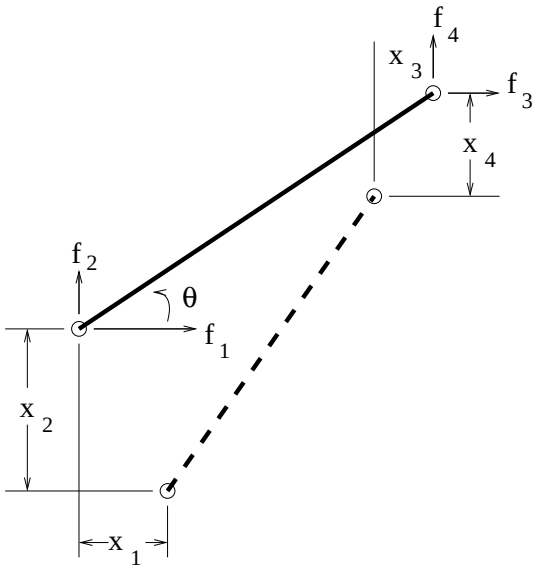


Figure 10.1. The reference (solid) and deformed (dashed) fiber.

Our goal is to build a theory that predicts x from knowledge of f. The first step is to quantify the associated **elongation**, $e \equiv \ell - L$, where L is the undeformed length and ℓ is the deformed length. With respect to Figure 10.1, we suppose that the lower left node of the undeformed fiber sits at $(0, 0)$ in the Cartesian plane, while its upper right node resides at $(L\cos\theta, L\sin\theta)$. Following Euclid, we write

$$\begin{aligned}\ell &= \sqrt{(L\cos\theta + x_3 - x_1)^2 + (L\sin\theta + x_4 - x_2)^2} \\ &= \sqrt{L^2 + 2L\{(x_3 - x_1)\cos\theta + (x_4 - x_2)\sin\theta\} + (x_3 - x_1)^2 + (x_4 - x_2)^2} \\ &= L\sqrt{1 + 2\{(x_3 - x_1)\cos\theta + (x_4 - x_2)\sin\theta\}/L + \{(x_3 - x_1)^2 + (x_4 - x_2)^2\}/L^2}.\end{aligned}$$

To help us here we invoke MacLaurin, $\sqrt{1+t} = 1 + t/2 + O(t^2)$ for small t, and write

$$\ell = L + (x_3 - x_1)\cos\theta + (x_4 - x_2)\sin\theta + O((x_j - x_{j+2})^2/L).$$

Hence, assuming that $(x_j - x_{j+2})^2$ is small (for both $j = 1$ and 2) compared with L, we find

$$\boxed{e \approx (x_3 - x_1)\cos\theta + (x_4 - x_2)\sin\theta.} \tag{1}$$

In the future we shall write $=$ instead of $\approx$. It will be understood that we are working under the hypothesis that the end displacements are small in comparison to the undeformed length. This approximation can be interpreted geometrically, or visually, to mean simply that each fiber tends to deform along its original direction.

We assume that our fiber is Hookean, in the sense that its restoring force, y, is proportional to its elongation. More precisely, we presume that

$$y = \frac{Ea}{L}e \tag{2}$$

where E denotes the fiber's Young's modulus, a denotes its cross sectional area, and L denotes its reference length. This y, positive when the bar is stretched and negative when compressed, acts along the reference direction, θ, in balance with the applied load f. More precisely, at the lower node

$$y\cos\theta + f_1 = 0 \quad \text{and} \quad y\sin\theta + f_2 = 0 \tag{3}$$

and at the upper node

$$y\cos(\pi+\theta) + f_3 = 0 \quad \text{and} \quad y\sin(\pi+\theta) + f_4 = 0$$

or

$$-y\cos(\theta) + f_3 = 0 \quad \text{and} \quad -y\sin(\theta) + f_4 = 0 \tag{4}$$

Finally, we need only substitute our expression for y in terms of e and e in terms of x and recognize that the 4 equations in (3) and (4) (hopefully) determine x from f. More precisely, with $k \equiv Ea/L$, we find

$$\begin{aligned}
k\{(x_3-x_1)\cos\theta + (x_4-x_2)\sin\theta\}\cos\theta &= -f_1\\
k\{(x_3-x_1)\cos\theta + (x_4-x_2)\sin\theta\}\sin\theta &= -f_2\\
k\{(x_3-x_1)\cos\theta + (x_4-x_2)\sin\theta\}\cos\theta &= f_3\\
k\{(x_3-x_1)\cos\theta + (x_4-x_2)\sin\theta\}\sin\theta &= f_4
\end{aligned}$$

Let us consider a few concrete examples. If

$$k=1,\ \theta=0,\ f_2=f_4=0 \quad \text{and} \quad f_1=-f_3$$

the above system of four equations deflates to

$$x_3 - x_1 = f_3$$

which indeed determines x_1 and x_3 up to an arbitrary rigid motion, stemming from the fact that our fiber is a floater.

Example 1. We nail things to a foundation, and add a fiber and arrive at the tent below.

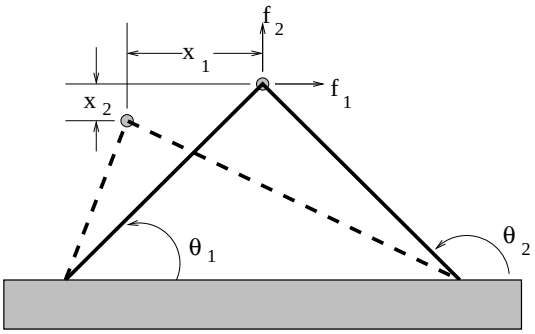


Figure 10.2. A simple tent.

We first compute the two elongations

$$e_1 = x_1 \cos(\theta_1) + x_2 \sin(\theta_1)$$
$$e_2 = x_1 \cos(\theta_2) + x_2 \sin(\theta_2)$$

we suppose the fibers have stiffnesses k_1 and k_2 and so

$$y_1 = k_1 e_1$$
$$y_2 = k_2 e_2$$

while force balance at the only free node yields

$$-y_1 \cos(\theta_1) - y_2 \cos(\theta_2) + f_1 = 0$$
$$-y_1 \sin(\theta_1) - y_2 \sin(\theta_2) + f_2 = 0$$

Assuming $\theta_1 = \pi/4$ and $\theta_2 = 3\pi/4$, we find

$$e_1 = (x_1 + x_2)/\sqrt{2}$$
$$e_2 = (x_2 - x_1)/\sqrt{2}$$

and

$$(y_1 - y_2)/\sqrt{2} = f_1$$
$$(y_1 + y_2)/\sqrt{2} = f_2$$

and so x must obey

$$(k_1 + k_2)x_1/2 + (k_1 - k_2)x_2/2 = f_1$$
$$(k_1 - k_2)x_1/2 + (k_1 + k_2)x_1/2 = f_2$$

In the case of bars of equal stiffness we find the simple answer that

$$x_1 = f_1/k_1 \quad \text{and} \quad x_2 = f_2/k_2.$$

For unequal stiffnesses we must solve our 2 linear equations simultaneously. Matlab knows Gaussian Elimination.

We are interested in understanding big nets (say m fibers meeting at n joints) and so we step back and realize that our model was constructed in three easy pieces.

The fiber elongations are linear combinations of their end displacements,

$$e = Ax,$$

where A is m-by-$2n$, is called the node-edge adjacency matrix, and encodes the 'geometry' of the fiber net.

Each fiber restoring force is proportional to its elongation,

$$y = Ke,$$

where K is m-by-m and diagonal and encodes the 'physics' of the fiber net.

The restoring forces balance the applied forces at each node,

$$A^T y = f$$

where A^T is the transpose (exchange rows for columns) of A.

When these steps are combined, we arrive at the linear system

$$A^T K A x = f.$$

For the net of Figure 10.2, we have

$$A = \begin{pmatrix} \cos\theta_1 & \sin\theta_1 \\ \cos\theta_2 & \sin\theta_2 \end{pmatrix} \quad K = \begin{pmatrix} k_1 & 0 \\ 0 & k_2 \end{pmatrix} \quad A^T = \begin{pmatrix} \cos\theta_1 & \cos\theta_2 \\ \sin\theta_1 & \sin\theta_2 \end{pmatrix}$$

We have coded this tent example here.

Example 2. We now build the adjacency matrix for the trailer below.

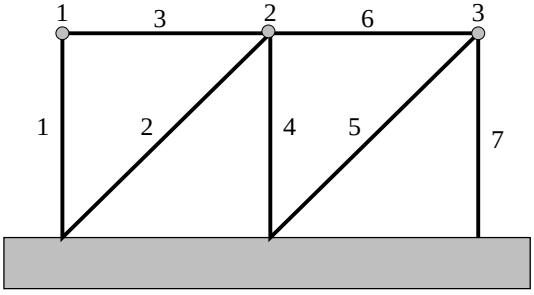


Figure 10.3. A trailer.

We have numbered the 7 fibers and the 3 nodes. We shall adopt the convention that the horizontal and vertical displacements of node j are x_{2j-1} and x_{2j} respectively. With the fiber angles,

$$\theta_1 = \theta_4 = \theta_7 = \pi/2, \quad \theta_2 = \theta_5 = \pi/4, \quad \text{and} \quad \theta_3 = \theta_6 = 0,$$

the associated elongations are

$$\begin{aligned} e_1 &= x_2 \\ e_2 &= (x_3 + x_4)/\sqrt{2} \\ e_3 &= x_3 - x_1 \\ e_4 &= x_4 \\ e_5 &= (x_5 + x_6)/\sqrt{2} \\ e_6 &= x_5 - x_3 \\ e_7 &= x_6. \end{aligned}$$

which we translate, 1 row, i.e., one fiber, at a time.

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & s & s & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & s & s \\ 0 & 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \qquad s = 1/\sqrt{2}.$$

We have coded this trailer example here. We next animate this scene by applying an incremental force in our Moving Trailer Code

Project: Plaza Castilla Elastica

You will load, solve and animate a class of cantilevers like that exhibited below. This one is 4 stories tall and so has 4x2=8 nodes (and so 2x8=16 degrees of freedom) and 4x4=16 fibers. The odd numbered fibers make an angle of $\pi/4$ with the horizontal. You will write a function

```
function cantilever(Ea,F,nof)
```

where Ea is the 4xnos-by-1 vector of modulus area products, F is the strength of the downward force at the upper right tip, and nof is the number of frames in the resulting collage. We have denoted the number of stories by nos.

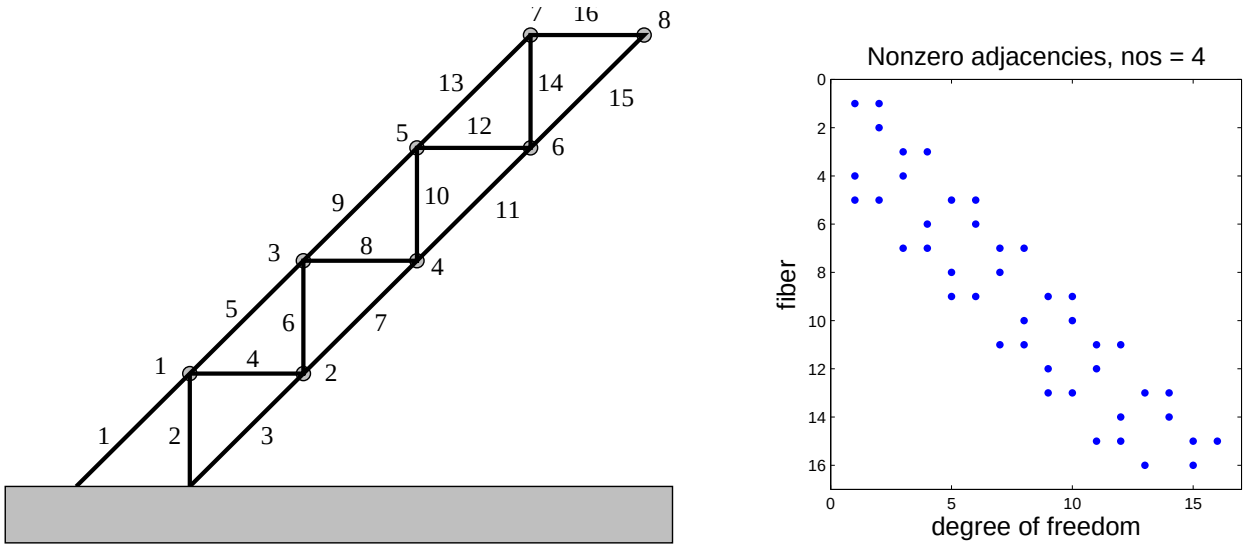


Figure 10.4. A cantilever and its adjacency matrix.

If you follow my ordering of the nodes and fibers then your adjacency matrix will exhibit considerable structure. This structure is revealed through Matlab's **spy** command as in the figure at the upper right.

Your code must accomodate an arbitrary number of stories and it must produce 2 figures. The first figure should come from spy (like that at the upper right) while the other should be a likeness of the Duchampian collage below. This figure is obtained by scaling the vertical load by frame number, as in our moving trailer example. I used one call to `line` to draw the net at each load and I used `fill` to draw the base.

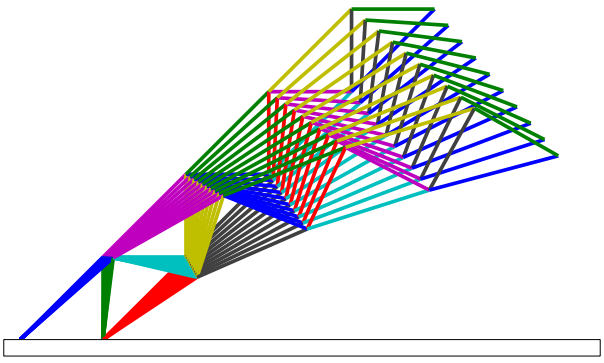


Figure 10.5. A descending cantilever.

Your work will be graded as follows:

```
6 pts for header CONTAINING detailed USAGE
8 pts for further comments in code
4 pts for indentation
8 pts for correct undeformed coordinates
8 pts for correct adjacency matrix
8 pts for correct plotting of deformed cantilever

4 pts for labeled plot of adjacency matrix for nos = 6
4 pts for plot of Duchampian cantilever for nos = 6
```

11. Gaussian Elimination

Looking back at our fiber code we see that all of the 'math' is hidden under the innocuous backslash - namely `x=S\f`. We strive in this subsection to reveal the hidden work by examining systems of equations of increasing complexity.

The easiest systems to solve are the uncoupled, or **diagonal** systems, i.e., systems of the form `Sx = f` where only the diagonal elements of `S` are nonzero. If `S` is n-by-n we simply write

```
for j=1:n
    x(j) = f(j)/S(j,j);
end
```

or, even simpler,

```
x = f./diag(S)
```

Of course this procedure breaks down if `S` has a zero on its diagonal. A diagonal matrix with one or more zeros on its diagonal is said to be **singular** . In such a case, `Sx=f` is not solvable unless `f` has a corresponding zero. Although, even in this case we note that `x` is left undetermined. For example

$$\begin{pmatrix} 2 & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} x(1) \\ x(2) \end{pmatrix} = \begin{pmatrix} 4 \\ 0 \end{pmatrix}$$

determines $x(1)$ but not $x(2)$.

After diagonal matrices the next most easily handled type is the class of **triangular** matrices. These are matrices all of whose nonzeros lie either on and above (**upper**) or below (**lower**). In the following upper triangular system

$$\begin{pmatrix} 4 & 2 \\ 0 & 3 \end{pmatrix} \begin{pmatrix} x(1) \\ x(2) \end{pmatrix} = \begin{pmatrix} 4 \\ 6 \end{pmatrix}$$

we start at the bottom and notice that $x(2) = 6/3 = 2$. The first equation then reads $4x(1) + 2x(2) = 4$ or $4x(1) = 4 - 2x(2)$ or $x(1) = (4 - 2x(2))/4 = 0$. This procedure is commonly known as **backsubstition**. With respect to coding, it requires only slightly more effort than in the diagonal case.

```
x = zeros(n,1);
x(n) = f(n)/S(n,n);
for j=n-1:-1:1
```

```
        tmp = 0;
        for k=j+1:n
            tmp = tmp + S(j,k)*x(k);
        end
        x(j) = (f(j) - tmp)/S(j,j);
    end
```

As in the diagonal case we find ourselves dividing by each diagonal element of `S`. Hence, a triangular matrix with one or more zeros on its diagonal is said to be **singular** . Gaussian elimination brutally seeks to reduce a given linear system to a triangular linear system via repeated application of **two** elementary row operations. The first is **Row Swapping**, for example,

```
S([j k],:) = S([k j],:)
```

and the second is **Row Mixing**, for example

```
S(j,:) = S(j,:) + magicnumber*S(k,:)
```

In each column we **swap** in order to bubble up a big **pivot** then we repeatedly **mix** until the **pivot** has eliminated every element below the diagonal. The systematic use of these two utilities is diagrammed in the following pseudo code:

```
function x = gauss(S,f)
n = length(f);
S = [S | f]     Augment S with f
for k=1:n-1     k counts columns
    r = row number, larger than or equal to k,
        with largest value (in magnitude) in column k
    if this largest value is really small then warn the user
    swap row r and row k
    for j=k+1:n
        mix row k into row j in order to eliminate S(j,k)
    end
end
if S(n,n) is really small then warn the user
strip off the changed f, i.e., copy column n+1 of S onto f
x = trisolve(S,f)
return
```

Here is a small example where `S` and `f` are:

```
2   4   2          2
1   2   4   and    3
4   2   1          1
```

Here is S augmented with f:

```
2   4   2   2
1   2   4   3
4   2   1   1
```

rows 1 and 3 are swapped:

```
4   2   1   1
1   2   4   3
2   4   2   2
```

our first **pivot** is 4 and elimination occurs in column 1:

```
4  2     1     1
0  3/2   15/4  11/4
0  3     3/2   3/2
```

rows 2 and 3 are swapped:

```
4  2     1     1
0  3     3/2   3/2
0  3/2   15/4  11/4
```

our second **pivot** is 3 and elimination occurs in column 2:

```
4  2     1     1
0  3     3/2   3/2
0  0     3     2
```

our third and final **pivot** is also 3, our matrix is now triangular, and so we may trisolve:

```
x(3) = 2/3
3*x(2) + (3/2)*(2/3) = 3/2          so x(2) = 1/6
4*x(1) + 2*(1/6) + 1*(2/3) = 1      so x(1) = 0
```

In the case that you encounter a small (smaller than eps) pivot your warning message might look something like that found in my singular diary.

Project: Fiber Nets and Gaussian Elimination

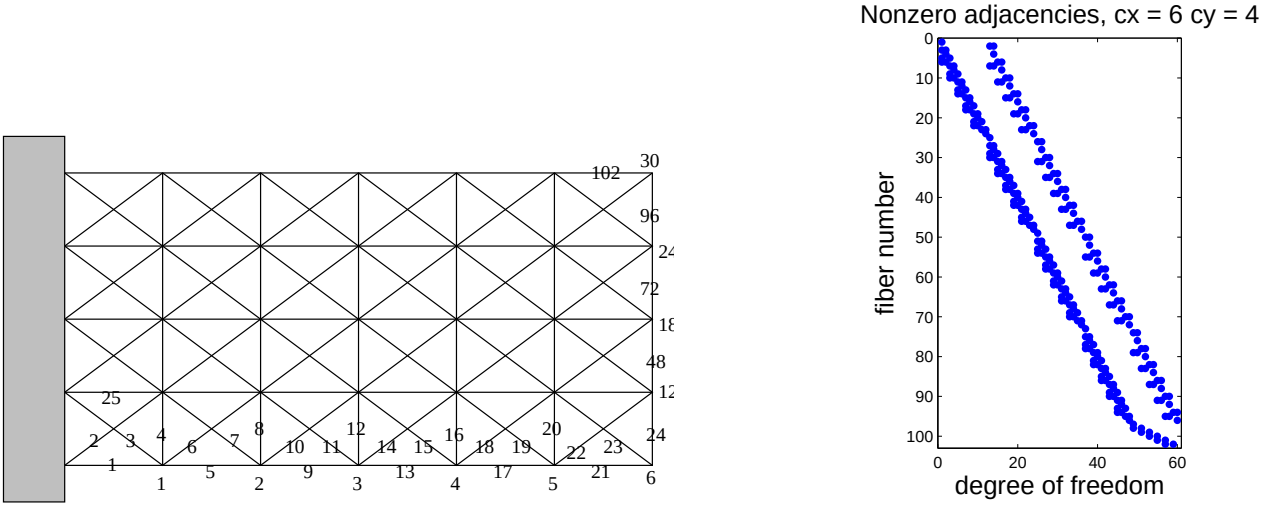


Figure 11.1. A cantilever and its adjacency matrix.

You will load, solve and animate a class of cantilevers like that exhibited above. This one is 6 cells wide and 4 cells tall. It has 102 fibers and 60 degrees of freedom. If the fibers and nodes are numbered as indicated then one arrives at an adjacency matrix like that spied above.

Regardless of the number of cells, this cantilever has length 2 and height 1. As a result, the angles of the cross bars will depend upon your choice of `cx` (the number of horizontal cells) and `cy` (the number of vertical cells). NOTE: The diagonal bars cross over one another without calling for a node.

To make sure we are all on the right track let us build the adjacency matrix by hand when $cx = cy = 2$. In this case, see fig. we have 18 fibers connected at 6 nodes and so the net enjoys 12 degrees of freedom.

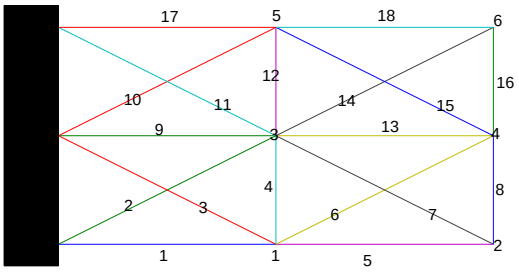


Figure 11.2. The cantilever with $cx = cy = 2$.

We assess their elongations via repeated reference to our general formula, equation (1) on page 44.

To begin, the elongation of the six horizontal fibers obey

$$e_1 = x_1, \quad e_5 = x_3 - x_1, \quad e_9 = x_5,$$
$$e_{13} = x_7 - x_5, \quad e_{17} = x_9, \quad e_{18} = x_{11} - x_9,$$

while the elongations of the vertical fibers obey

$$e_4 = x_6 - x_2, \quad e_8 = x_8 - x_4, \quad e_{12} = x_{10} - x_6, \quad e_{16} = x_{12} - x_8.$$

Regarding the diagonal fibers we must first compute the associated angles. The length of fiber one is $(2/cx) = 1$ while the length of fiber 4 is $1/cy = 1/2$ and so

$$\cos(\theta_1) = 2/\sqrt{5} \quad \text{and} \quad \sin(\theta_1) = 1/\sqrt{5}.$$

With $s \equiv 1/\sqrt{5}$ equation (1) now provides

$$e_2 = 2sx_5 + sx_6, \quad e_6 = 2s(x_7 - x_1) + s(x_8 - x_2),$$
$$e_{10} = 2sx_9 + sx_{10}, \quad e_{14} = 2s(x_{11} - x_5) + s(x_{12} - x_6).$$

Finally, as $\theta_2 = \pi - \theta_1$ we find that

$$\cos(\theta_2) = -2/\sqrt{5} \quad \text{and} \quad \sin(\theta_2) = 1/\sqrt{5}$$

and so,

$$e_3 = -2s(0 - x_1) + s(0 - x_2), \quad e_7 = -2s(x_5 - x_3) + s(x_6 - x_4),$$
$$e_{11} = -2s(0 - x_5) + s(0 - x_6), \quad e_{15} = -2s(x_9 - x_7) + s(x_{10} - x_8).$$

On gathering all 18 elongations we arrive at the adjacency matrix

$$A = \begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 2s & s & 0 & 0 & 0 & 0 & 0 & 0 \\
2s & -s & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
-1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-2s & -s & 0 & 0 & 0 & 0 & 2s & s & 0 & 0 & 0 & 0 \\
0 & 0 & 2s & -s & -2s & s & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2s & s & 0 & 0 \\
0 & 0 & 0 & 0 & 2s & -s & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -2s & -s & 0 & 0 & 0 & 0 & 2s & s \\
0 & 0 & 0 & 0 & 0 & 0 & 2s & -s & -2s & s & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0
\end{pmatrix}$$

I recommend that you test your code against these values.

You will adapt and extend last week's function to

```
function cantilever2(cx,cy,Ea,F,nof)
```

where **cx** and **cy** dictate the geometry of the net, Ea is the vector of modulus area products, F is the strength of the downward force at the center of the right face (node 18 in the figure above), and nof is the number of frames in the resulting movie. NOTE: The right face has no center unless cy is even. Your code may presume that the user will only call it with an even cy.

In addition to encoding a new net this week's code must also replace last week's miraculous `x=S\f` with

```
x = gauss(S,f)
```

where `gauss` is a subfunction that you must write and append to `cantilever2`. Recall that this subfunction has been sketched for you on the lecture page. In particular, it itself calls the subfunction `trisolve`. You should of course test your gauss against backslash on a number of random matrices before tucking it into cantilever2.

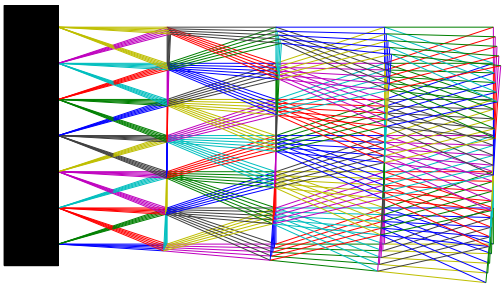


Figure 11.3. The bending of the cantilever.

Your code must accomodate an arbitrary number of cells and it must produce a figure via `spy(A)` like that above, and a Duchampian trace like that at right, of a net undergoing an incremental force at the center of its right face. (F = 0.05, nof = 5).

Your work will be graded as follows:

```
6 pts for header CONTAINING detailed USAGE
6 pts for further comments in code
4 pts for indentation
4 pts for correct undeformed coordinates
6 pts for correct adjacency matrix
6 pts for correct plotting of deformed cantilever
4 pts for correct row swapping in gauss
6 pts for correct row mixing in gauss

4 pts for labeled plot of adjacency matrix, cx=4, cy=6
4 pts for 5 frame plot of moving cantilever, cx=4, cy=6
```

12. Optimal Design of Fiber Nets

Biological fiber nets have (likely) evolved to best fit their niche. How one measures 'fitness' and how one identifies nature's 'design variables' are each matters of considerable debate. We will focus on the simplest such measure, namely compliance, or work done by the load, and suppose that nature may tune the radii of her fibers. Recalling that work is merely the product of force and distance we express compliance as

$$C = x^T f \tag{12.1}$$

where x is the solution to

$$A^T K(a) A x = f \tag{12.2}$$

where we have stressed the dependence of K, the elemental stiffness matrix, upon a, the vector of fiber cross sectional areas. We will presume throughout that each fiber has the same Young's modulus. Our interest then is in minimizing the C of (12.1) subject to x obeying the equilibrium equation (12.2). A moment's reflection will confirm that the compliance can be reduced to zero by simply choosing infinitely fat fibers. We preclude this possibility by constraining the volume of the net, via

$$L^T a = V \tag{12.3}$$

where L is the column vector of fiber lengths.

Let us solve a few small design problems by hand before opening the big toolbox. In particular, the displacement of the loaded symmetric ($\theta = \pi/4$) tent is

$$\begin{pmatrix} x(1) \\ x(2) \end{pmatrix} = \frac{s}{a(1)a(2)} \begin{pmatrix} a(1)+a(2) & a(2)-a(1) \\ a(2)-a(1) & a(1)+a(2) \end{pmatrix} \begin{pmatrix} f(1) \\ f(2) \end{pmatrix}$$

where $s^2 = 1/2$ as usual. And so

$$C = \frac{s(f(1)-f(2))^2}{a(2)} + \frac{s(f(1)+f(2))^2}{a(1)}$$

If we now invoke the volume constraint

$$a(1) + a(2) = sV$$

we find that

$$C = \frac{s(f(1)-f(2))^2}{sV - a(1)} + \frac{s(f(1)+f(2))^2}{a(1)}$$

and so we have reduced ourselves to a one-dimensional minimization problem. Let us consider a pair of special loadings.

Design against falling debris: In this case $f = [0;\ -1]$ and so

$$C = \frac{s}{a(1)} + \frac{s}{sV - a(1)} = \frac{s^2 V}{a(1)(sV - a(1))}$$

which attains its (positive) minimum at

$$a(1) = a(2) = sV/2.$$

This tells us that the two tent legs should be equally fat to best sustain a vertical load.

Design against bears: In this case $f = [1;\ 1]$ and so

$$C = \frac{4s}{a(1)}$$

which approaches its minimum as $a(1)$ approaches infinity. This of course leads to the absurdity that $a(2)$ approaches negative infinity. If we impose the additional constraints that $a(1)$ and $a(2)$ each be nonnegative then we arrive the optimal design

$$a(1) = sV, \quad a(2) = 0.$$

This corresponds to the fact that, as the load is perpendicular to the tent's right leg, the right leg does no work. In reality, in order to account for variations in the loading, one typically sets a nonzero lower bound on each $a(j)$, and for cost or construction contingencies one likewise also imposes a finite upper bound on each $a(j)$. In other words, we ask that

$$alo \le a(j) \le ahi \qquad \text{for each } j. \tag{12.4}$$

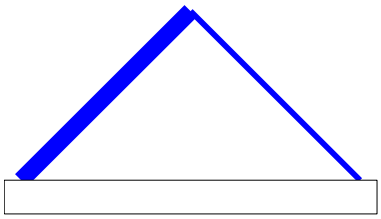


Figure 12.1. Best design against bears.

We now arrive at the **General Case**

$$\min_a x^T f$$

subject to

$$A^T K(a) A x = f, \quad L^T a = V \quad \text{and} \quad alo \le a(j) \le ahi \quad \text{for each } j.$$

MATLAB has a large toolbox for solving such **optimization** problems, type `help optim`. The routine most pertinent to our problem is `fmincon`. Its application to our simple tent is coded here. Running this code with $f = [.5; -1]$ reveals the figure at right.

As we are now moving into larger nets we should attempt to give `fmincon` a hand. `fmincon` searches for a minimum of the constrained compliance by searching for a critical point, that is an `a` for which the gradient, i.e., the vector of derivatives of the compliance with respect to `each` a(j), is zero. Hence, to accelerate `fmincon` we need only pass along information on the gradient of the compliance. The gradient we need is

$$\nabla C(a) = \left(\frac{\partial C(a)}{\partial a_1} \quad \frac{\partial C(a)}{\partial a_2} \quad \cdots \quad \frac{\partial C(a)}{\partial a_m} \right)^T.$$

Let's compute these one at a time, e.g.,

$$\frac{\partial C(a)}{\partial a_1} = f^T \frac{\partial x(a)}{\partial a_1}.$$

To compute the latter we differentiate the equilibrium equation

$$A^T K(a) A x(a) = f$$

with respect to a_1 and find, via the product rule and the fact that A and f are independent of a,

$$A^T \frac{\partial K(a)}{\partial a_1} A x(a) + A^T K(a) A \frac{\partial x(a)}{\partial a_1} = 0.$$

We solve this for

$$\frac{\partial x(a)}{\partial a_1} = -(A^T K(a) A)^{-1} A^T \frac{\partial K(a)}{\partial a_1} A x(a)$$

and so arrive at the desired

$$\begin{aligned} f^T \frac{\partial x(a)}{\partial a_1} &= -x^T(a)(A^T K(a) A)(A^T K(a) A)^{-1} A^T \frac{\partial K(a)}{\partial a_1} A x(a) \\ &= -(Ax(a))^T \frac{\partial K(a)}{\partial a_1} A x(a) \\ &= -(e_1 \; e_2 \; \cdots \; e_m) \begin{pmatrix} 1/L_1 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & \cdots & 0 \end{pmatrix} \begin{pmatrix} e_1 \\ e_2 \\ \vdots \\ e_m \end{pmatrix} \\ &= -e_1^2(a)/L_1. \end{aligned}$$

Arguing in a similar fashion for the remaining partial derivatives we find

$$\nabla C(a) = -(e_1^2(a)/L_1 \quad e_2^2(a)/L_2 \quad \cdots \quad e_m^2(a)/L_m)^T.$$

Do you see how minimizing the compliance is synonymous with minimizing the square of each elongation?

Project: Optimal Design of Fiber Nets

We turn to `fmincon` to design the strongest possible cantilever of volume `V` with the option of freezing the size of the horizontal fibers.

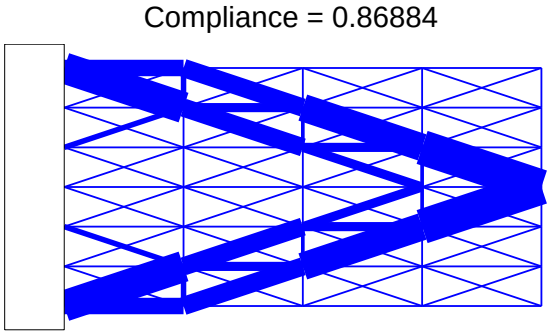


Figure 12.2. The best cantilever in its class.

You will write a function

```
function cantilever3(cx,cy,F,V,nohor)
```

that builds last week's adjacency matrix, `A`, vector of fiber lengths, `L`, and force vector, `f`. This `f` should be all zeros except for the value `F` in the component corresponding to vertical force at the center of the right face. (There is no center if cy is not even). Your cantilever3 will also generate an initial guess for the fiber cross-sectional areas, `a0`, that jibes with the volume constraint, `L'*a0 = V`. Your bestcant will then set lower and upper bounds on the permissible `a` and then call `fmincon` to find the best `a`. `fmincon` will in turn call your `compcant` function, where the compliance of the fiber net is computed. Each of these steps is a straighforward generalization of our best tent code.

You will inform `fmincon` that you are supplying the gradient of your objective function by switching on one of its options and by coding `compcant` to return both the compliance (1-by-1) and its gradient (m-by-1). In particular, you must follow this protocol

```
function [comp,grad] = compcant(a,A,L,f)
```

as in the simple example.

Once `fmincon` returns the best `a` it should be plotted, as above. For visual purposes you may wish to use the fat-factor-5 of btent.

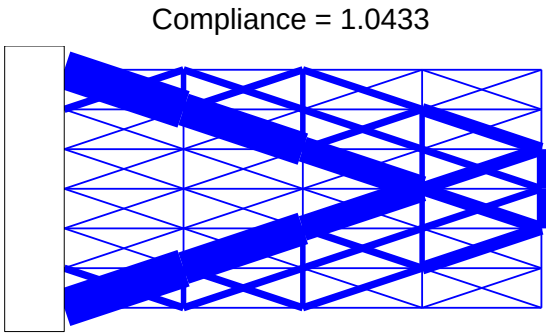


Figure 12.3. The best cantilever in the class of cantilevers without horizontal stiffeners.

Finally, you will write a simple driver

```
function cant3drive
```

that sets cx, cy, F and V as well as specifies whether or not all of the horizontal fiber areas should be held at their LB values (if nohor=1 then UB=LB=0.001 for each horizontal fiber). With nohor=0 you should achieve the figure above. With nohor=1 you should achieve the figure at right.

Your work will be graded as follows:

```
6 pts for header CONTAINING detailed USAGE
8 pts for further comments in code
4 pts for indentation
8 pts for correct cantilever3
8 pts for correct compcant with gradient

with cx = 4, cy = 6, V = 20 and  F = -0.5

8 pts for labeled plot of best net, nohor=0
8 pts for labeled plot of best net, nohor=1
```

13. Multi-Layer Perceptrons

As our next example we shall study networks of simple minded neurons. Attractors on these nets will look and behave much like the attractors for our gene nets. Our interest however is not the behavior of a single net but rather a net's capacity to learn in both supervised and unsupervised settings.

This is not so far from your work on optimal fiber nets. In that case you began with a random set of fiber diameters and a given load and the net slowly **learned** how to redistribute its volume in such a way as to minimize the work done by the load. The role of **teacher** was played by `fmincon`.

In this setting we shall teach a neural network to respond in a desired way when faced with certain stimuli.

To learn is to modify one's synaptic weights.

Let us start with a simple example. One neuron with two inputs. The neuron scales and sums its inputs and fires if the sum exceeds a given threshold. If the inputs and output are binary and the threshold for firing is 0.5 then

```
out = round(w(1)*in(1) + w(2)*in(2))
```

is our neural net. The challenge is to choose `w1` and `w2` in accordance with a particular task. For example, can we teach this neuron to perform the **and** operation? More precisely, we search for `w1` and `w2` such that

```
round(w(1)*0 + w(2)*0) = 0
round(w(1)*0 + w(2)*1) = 0
round(w(1)*1 + w(2)*0) = 0
round(w(1)*1 + w(2)*1) = 1
```

Here is one possible approach. You see that `w` learns to associate each stimulus with its preferred target by following the negative gradient of the error.

This example is of course a few hundred billion cells short of a full brain and is perhaps a little abrupt with respect to threshold. Let us consider bigger more gentle networks. We will consider networks with 3 layers comprised of

```
n   inputs
h   hidden cells, and
m   outputs
```

The h-by-n weighting matrix from inputs to hidden cells will be called `V` and the m-by-h weighting matrix from hidden cells to outputs will be called `W`. We offer a concrete example in the figure below.

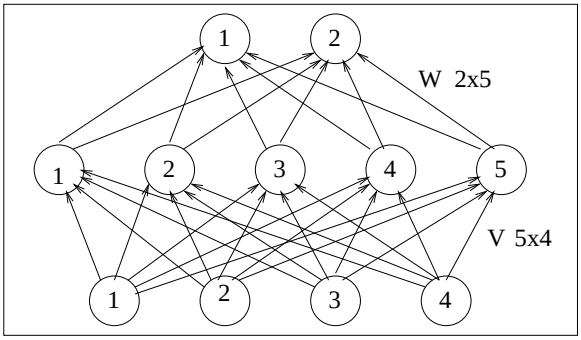


Figure 13.1. A three–layer Multiperceptron.

The hard threshold at each cell will be replaced by the soft sigmoid

```
s(x) = 1./(1+exp(0.5-x))
```

With this notation, if `p` is an input pattern then the output will be

```
o = s(Ws(Vp))
```

This representation is dangerously brief - please make sure you understand it before proceeding.

We will typically have `N` input patterns that we lay into the columns of the n-by-N matrix

```
p(:,1), p(:,2), ..., p(:,N)
```

that we would like to pair with `N` targets that comprise the columns of the m-by-N matrix

```
t(:,1), t(:,2), ..., t(:,N).
```

We do this by choosing `V` and `W` to minimize the misfit

$$\begin{aligned} E(V,W) &= \frac{1}{2}\sum_{i=1}^{N}(o(:,i)-t(:,i))^T(o(:,i)-t(:,i)) \\ &= \frac{1}{2}\sum_{i=1}^{N}(s(Ws(Vp(:,i)))-t(:,i))^T(s(Ws(Vp(:,i)))-t(:,i)) \\ &= \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{m}(s(W(j,:)s(Vp(:,i)))-t(j,i))^2. \end{aligned}$$

We minimize this sum of squares by guiding V and W along the gradient of E. That is, we update V and W according to

$$W = W - r\text{grad}_W E$$
$$V = V - r\text{grad}_V E$$

for some fixed learning rate, r. These derivatives are easiest to see if there is but one input pattern (N=1) and a single output cell (m=1), for then the misfit is simply

$$E(V,W) = (1/2)(s(Ws(Vp)) - t)^2$$

and so the chain rule reveals the gradient, or vector of partial derivatives

$$\text{grad}_W E(V,W) = [\partial E/\partial W_1 \ \partial E/\partial W_2 \ \cdots \ \partial E/\partial W_h]$$

takes the form

```
grad_W E(V,W) = (s(Ws(Vp))-t) * s'(Ws(Vp)) * s(Vp)^T
                      1x1            1x1       1xh
```

And similarly, the matrix of partial derivatives,

$$\text{grad}_V E(V,W) = \begin{pmatrix} \partial E/\partial V_{1,1} & \partial E/\partial V_{1,2} & \cdots & \partial E/\partial V_{1,n} \\ \partial E/\partial V_{2,1} & \partial E/\partial V_{2,2} & \cdots & \partial E/\partial V_{2,n} \\ \cdots & \cdots & \cdots & \cdots \\ \partial E/\partial V_{h,1} & \partial E/\partial V_{h,2} & \cdots & \partial E/\partial V_{h,n} \end{pmatrix}$$

takes the form

```
grad_V E(V,W) = (s(Ws(Vp))-t)*s'(Ws(Vp))*(W^T.*s'(Vp))*p^T
                      1x1        1x1          hx1        1xn
```

These expressions can be simplified a bit upon observing that our sigmoid obeys

```
s'(x) = s(x)(1-s(x))
```

and so, setting

```
q = s(Vp)  and  o = s(Wq)  brings
grad_W E = (o-t)o(1-o)q^T  and
grad_V E = (o-t)o(1-o)(W^T.*q.*(1-q))p^T
```

We have coded this training here. We test the net it generates using nnxor, in this diary

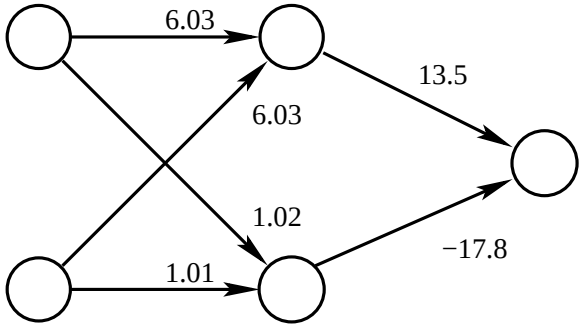


Figure 13.2. A multiperceptron that performs exclusive OR.

```
>> [V,W] = xortrain([5 5;1 1],[10 -15],10000,0.25)
V = 6.0321    6.0311
    1.0216    1.0066
W = 13.4652  -17.7826

>> nnxor([0 0]',V,W)
ans = 0.1062

>> nnxor([0 1]',V,W)
ans = 0.8600

>> nnxor([1 0]',V,W)
ans = 0.8524

>> nnxor([1 1]',V,W)
ans = 0.1615
```

Do you see that our pupil has stumbled upon the Boolean Identity

```
XOR(a,b) = OR(a,b) - AND(a,b)
```

You see from the code that we have proceeded as if there was only one input pattern by simply choosing one at random each time. The same ploy may be used in the case of multiple outputs, as, e.g., in this week's assignment.

Project: - DNA OCR via Neural Nets

Train a 3 layer neural net to recognize digitized versions of the letters A, C, G and T. Your net should have 25 inputs, 2 outputs and 25 "hidden" cells. The inputs will be binary and correspond to the LED-like figure below.

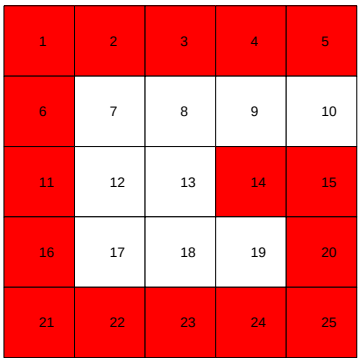


Figure 13.3. The input pattern [1 1 1 1 1 1 0 0 0 0 1 0 0 1 1 1 0 0 0 1 1 1 1 1 1].

For your targets, please use

```
T = [0 0],  A = [0 1],  C = [1 0],  and G = [1 1].
```

Write a function

```
[V,W] = ocrtrain(V0,W0,maxiter,rate)
```

that returns the two weight matrices after successful training at the given rate.

Drive this function with a function called ocrdrive that

```
sets maxiter=5000 and rate=0.1,
sets initial weights V0 and W0 via randn,
sends all this info to ocrtrain and receives
      the trained V and W,
displays V and W as below via imagesc, colorbar,
      hist, reshape, and subplot
applies these learned V and W to a 12-by-25
      bitstream whose 1st four rows are
[1 1 1 1 1 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 1 1 1 1;
 1 1 1 1 1 1 0 0 0 0 1 0 0 1 1 1 0 0 0 1 1 1 1 1 1;
 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1 0 0 0 1 1 0 0 0 1;
 1 1 1 1 1 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0]
to get the next 4 four rows flip 1 bit in each of the above,
to get the final 4 four rows flip 2 bits in each of the above.
Displays these 12 letters as below via, subplot,
   imagesc, reshape and colormap(1-gray),
Finally, display in the command window the sequence of letters
   that V and W determined were represented in the bitstream,
   e.g., seq = CGATCGATCGAA
```

Hint The grad formulas established above work nice for scalar outputs. But here your outputs have two components, in particular your W has two rows.

What to do? Well, on each iteration just work on updating a single row of W. How should you choose which row? Flip a `rand`. **2nd Hint:** To get exactly my V and W we should start at the same random $V0$ and $W0$. To do that please reinitialize `randn` before each use via `randn('state',0)`.

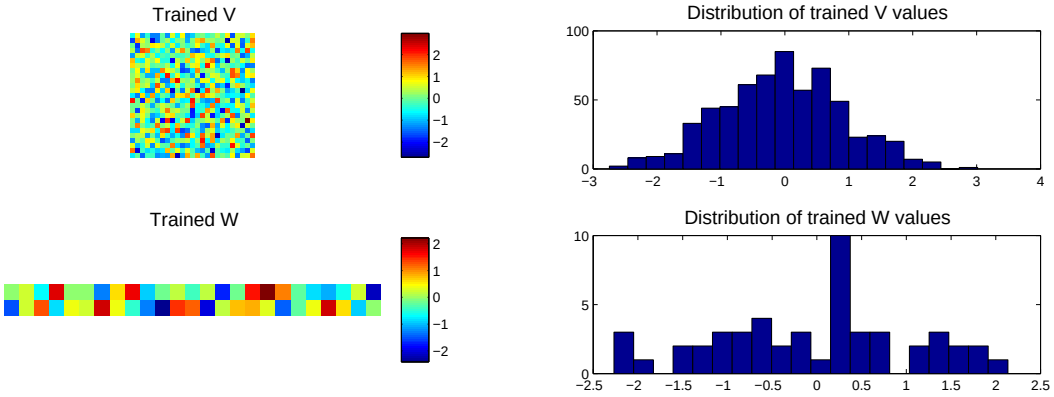


Figure 13.4. Representations of the trained weight matrices.

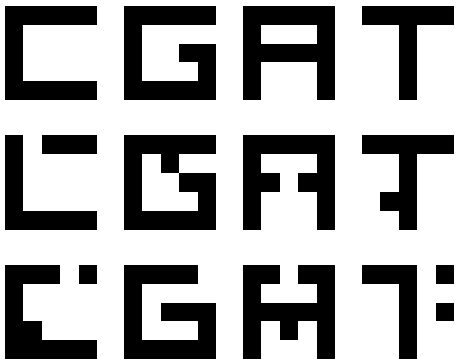


Figure 13.5. Progressively noisier input patterns.

Your work will be graded as follows

```
8 pts for headers CONTAINING detailed USAGE
8 pts for further comments in code
4 pts for indentation
10 pts for correct ocrtrain
8 pts for correct ocrdrive
4 pts for V and W titled imagesc plots
4 pts for V and W titled hist plots
4 pts for display of final letters to command window
```

14. Hopfield Nets

Hopfield nets are comprised of N hard-threshold nodes with all-to-all symmetric coupling where on = 1 and off = -1. Let us start with the net below

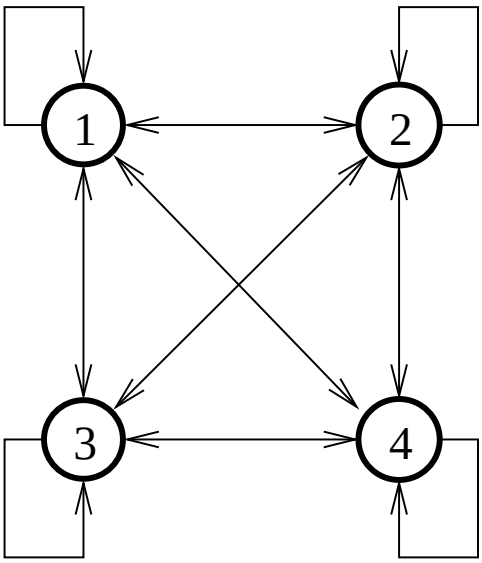


Figure 14.1. A four node Hopfield net.

There are 4 nodes and so we must build a 4-by-4 weight matrix W. The bidirectional arrows imply equal weights in each direction, i.e.,

$$W(i,j) = W(j,i)$$

If s is the current state of the net then the next state is

$$ns = \text{sign2}(Ws)$$

where

$$\text{sign2}(x) = \begin{cases} 1 & \text{if} \quad x > 0 \\ -1 & \text{if} \quad x \leq 0 \end{cases}$$

is our hard threshold function. This net can be trained to remember an input pattern p by setting the weights to

$$W = pp^T$$

then p and its mirror image $-p$ will be **attractors**. That is, they will satisfy $s = \text{sign2}(Ws)$. To see this note that (as $p^Tp > 0$)

$$Wp = pp^Tp = p(p^Tp) = (p^Tp)p \quad \text{and so} \quad \text{sign2}(Wp) = \text{sign2}(p) = p.$$

In fact, for **each** initial state, s, the very next state will be either $\pm p$ or `-ones(N,1)` (when $p^T s = 0$), for

$$Ws = pp^T s = p(p^T s) = (p^T s)p \quad \text{and so} \quad \text{sign2}(Ws) = \text{sign2}(p^T s)\text{sign2}(p),$$

unless p is orthogonal to s, i.e., $p^T s = 0$, in which case

$$\text{sign2}(Ws) = -\text{ones}(\text{N}, 1).$$

If this full off vector is itself orthogonal to p then it and $\pm p$ will be the only attractors. To get a feel for this I encourage you to dissect and exercise hop on several 4-by-1 choices.

All of this generalizes nicely to multiple training patterns. In fact, if $p(:,1)$ and $p(:,2)$ are two such patterns we set

$$P = [p(:,1)\ p(:,2)] \quad \text{and} \quad W = PP^T$$

Arguing as above, we find

$$Ws = PP^T s = (s^T p(:,1))p(:,1) + (s^T p(:,2))p(:,2)$$

Evalutating sign2 of this is now a much more interesting affair. If $p(:,1)$ and $p(:,2)$ are orthogonal to one another, i.e., $p(:,1)^T p(:,2) = 0$ then it is not hard to see that both $p(:,1)$ and $p(:,2)$ (and their mirrors) will be attractors. In the nonorthogonal case we must consider the elements of $V = P^T P$. To get a feel for this I encourage you to exercise hop on several 4-by-2 choices. For example, with

$$P = \begin{pmatrix} 1 & 1 \\ 1 & 1 \\ -1 & 1 \\ -1 & -1 \end{pmatrix}$$

we find

$$V = \begin{pmatrix} p(:,1)^T p(:,1) & p(:,1)^T p(:,2) \\ p(:,1)^T p(:,2) & p(:,2)^T p(:,2) \end{pmatrix} = \begin{pmatrix} 4 & 2 \\ 2 & 4 \end{pmatrix}$$

and so $p(:,1)$ and $p(:,2)$ (and their mirrors) are attactors. In fact they are the only attractors. A little experimentation reveals that

```
[+ + - +] & [+ - - -]                          -> [+ + - -]
[+ - + +], [- + + +], [+ - - -] & [- - + -] -> [- - + +]
[+ - + -], [- + + -] & [+ + + +]               -> [+ + + -]
[- - - -], [+ - - +] & [- + - +]               -> [- - - +]
```

Project: Face Discrimination via Hopfield

We will apply Hopfield technology to a variation of last week's task. In particular, we will write a function, `hopoli`, that

1. Reads and shows the four images, Obama, Clinton, McCain, and Palin and converts each to bitstreams of ± 1 and displays each as a training pattern as in Fig. 14.2.
2. Constructs and displays the associated Hopfield weight matrix W (as in Fig. 14.2) and (iteratively) applies it to very (three fourths of the pixels are random) noisy images of our 4 candidates and displays the before and after as in Figs. 14.3 and 4.

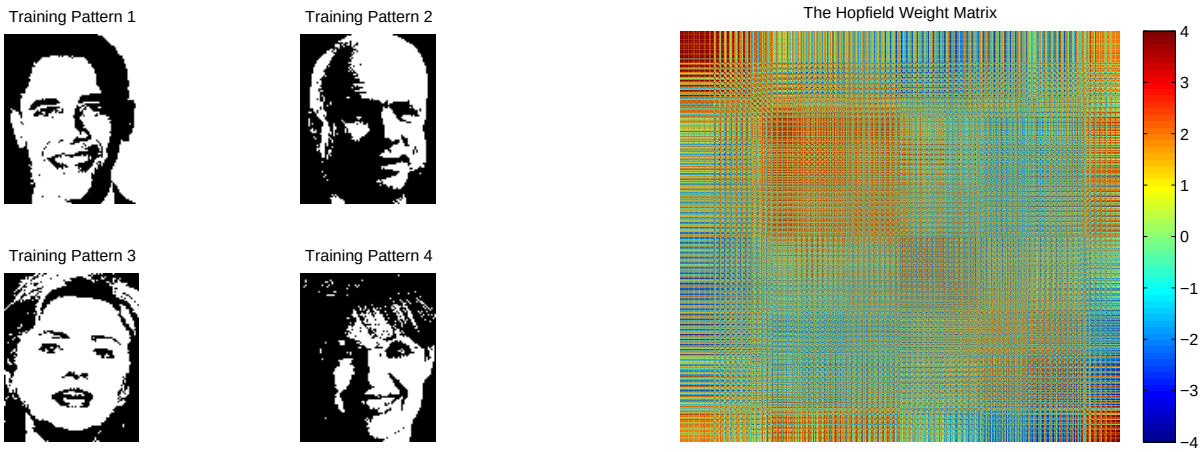


Figure 14.2. The black and white candidates and their associated Hopfield weight matrix.

Figure 14.3. The men pulled from the noise.

Figure 14.4. The women pulled from the noise.

We will use MATLAB's Image Processing Toolbox to read, transform, and show these images. In particular,

```
a = imread('obama.jpg');           % read the jpg
abw = im2bw(a);                 % convert to black and white
abwc = abw(25:435,60:end-60);      % crop it
abwcc = abwc(1:4:end,1:4:end);     % coarsen it
imshow(abwcc)                      % show it
```

achieves the first training pattern. This 'final' matrix is now 103-by-82 and is of type 'logical' (i.e., T/F or 1/0). To get your bitstream you must `reshape` this into a vector of length 103*82=8446. You must also turn the zeros into -ones. I recommend using `real`. By similar means you can binarize, crop and coarsen the other portraits into bitstreams of length 8446. NOTE: This is a fairly ambitious bitstream, for it produces a weight matrix W that is 8446-by-8446. As MATLAB requires more that 570 megabytes to store this I fear that, depending on your hardware, you may receive a memory error. If that is the case simply coarsen your image until W fits.

Your work will be graded as follows:

```
6 pts for headers CONTAINING detailed USAGE
4 pts for further comments in code
4 pts for indentation
11 pts for correct W
11 pts for correct application of W
14 pts for labeled figures (as above)
```

15. Evolutionary Game Theory

Please visit the Stanford Encyclopedia of Philosophy for an overview of the theory. We will proceed, as usual, by example. In particular, we will play **Prisoner's Dilemma** and the **Game of Life** in preparation for combination of these games in your invesitigation of the **Evolution of Cooperation**.

Prisoner's Dilemma

Two freshmen have been arrested by RUPD for commandeering a golf cart and fleeing an officer. They are placed in separate isolation cells. The wily Dean makes the following offer to each.

> "You may choose to confess or remain silent. If you confess and your accomplice remains silent I will drop all charges against you and charge your accomplice with felony theft. Likewise, if your accomplice confesses while you remain silent, he will go free while you will be charged with felony theft. If you both confess I get two convictions, but I'll see to it that you both receive early parole. If you both remain silent, I'll have to settle for charging you both with the lesser charge of avoiding arrest. If you wish to confess, you must leave a note with the jailer before my return."

The dilemma is that it pays to confess, so long as your accomplice does not also confess.

This game is abstracted as follows. To remain silent is to cooperate with your accomplice and so this strategy is denoted by the letter C. To confess is to break with, or defect from, your accomplice and so this stategy is denoted by the letter D. The payoffs to the row player during a round are

```
    C   D          R is the reward for mutual cooperation
C   R   S   where  P is the punishment for mutual defection
D   T   P          T is the temptation to defect and
                   S is for sucker

and  T > R > P > S  ensures dilemma
```

In this week's assignment you will conduct such games over multiple generations and between many many players. In preparation for the incorporation of multiple players let us play the

Game of Life

The game is played on an M-by-N board, where the living are ones and the dead are zeros, and the rules are brutally simple.

```
Any live square with fewer than 2, or more than 3,
    neighbors dies.
Any dead square with exactly 3 neighbors
    comes to life.
```

Please dissect and exercise the associated code.

Project: Evolution of Cooperation

We will reproduce the findings of May and Nowak on Evolutionary Games and Spatial Chaos. The game takes place on an M-by-N board. Each square is occupied by a cooperator (C) or a defector (D).

In a given round each square plays prisoner's dilemma with each of its immediate neighbors, including itself. Each square has either 4, 6, or 9 such neighbors. The payoffs are as follows

```
C vs. C, each receives 1
C vs. D, C receives 0 and D receives b
D vs. D, each receives 0
```

The **score** of a square is the sum of the neighborly payoffs.

At the next generation, each square assumes the identity of the winner (highest scorer) of the last neighborly round.

For example, the score of the board

```
D  D  D          b 2b  2b
D  C  C   is    3b  5  4
C  C  C          3  5  4
```

and so if b=1.9 the next generation is

```
D  D  C
D  D  C
D  D  C
```

We monitor the game's progress by

```
painting a square blue when C remains C
painting a square red when D remains D
painting a square yellow when C becomes D
painting a square green when D becomes C
```

You will write the

```
function evo(M,N,b,gen)
```

that plays this game for **gen** generations, commencing from a board of cooperators with a lone defector at its center (when M and N are odd). With $M = N = 99$ and $b = 1.9$ you will find boards like

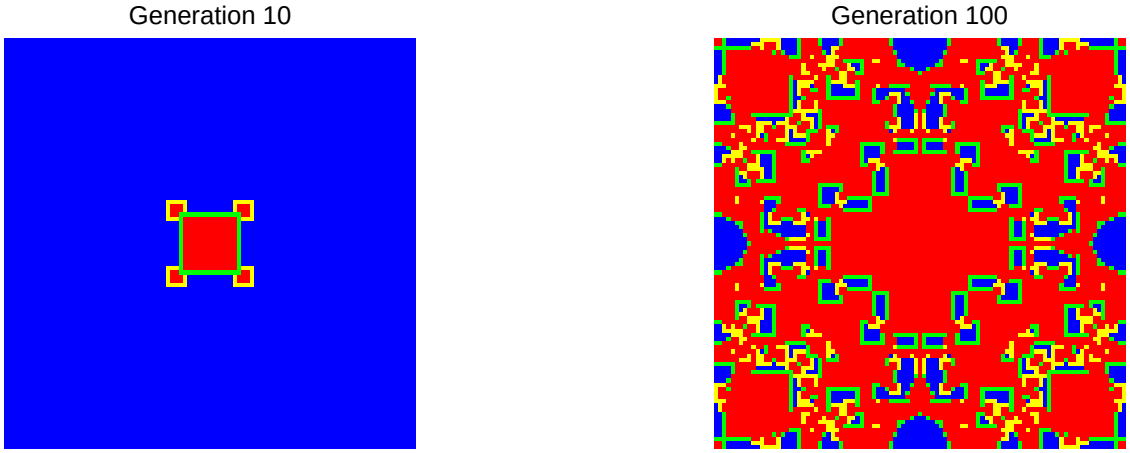


Figure 15.1. The boards at generation 10 and 100.

after 10 and 100 generations respectively. It is fascinating to watch the fraction of cooperators ebb and flow. You will want to quantify this in a graph of the form

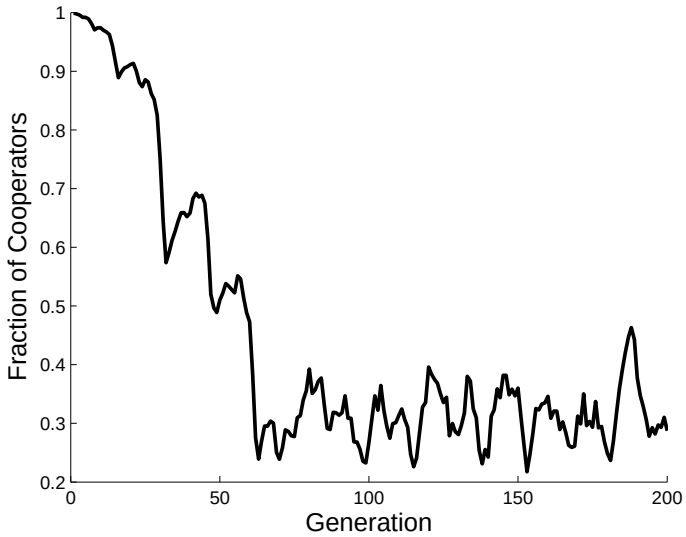


Figure 15.2. The cooperators take a hit but hold on.

I recommend that evo delegate to three subfunctions

```
function S = score(A,b)
```

```
function An = advance(S,A)
function evodisp(A,An)
```

where **A** denotes the state of the previous generation. The state is the matrix of identities (C or D, simlilar to live and dead) of each of the players.

Finally, construct a driver that sets sizes and b in order to generate the plots listed below.

Your work will be graded as follows

```
5 pts for headers CONTAINING detailed USAGE
10 pts for further comments in code
5 pts for indentation
20 pts for correct score function
20 pts for correct advance function
15 pts for correct evodisp function

  in the following, suppose  b=1.9,
  and start from the C board with lone center D

5 pts for fraction of cooperators plot when M=67, N=67 and gen = 40
5 pts for fraction of cooperators plot when M=65, N=69 and gen = 100
5 pts for fraction of cooperators plot when M=69, N=69 and gen = 200
5 pts for plot of game board at generation 100, when M=199, N=199
5 pts for plot of game board at generation 200, when M=199, N=199
```